What is LuaEdit?

LuaEdit is a script editor designed for the 5.0.2 version of Lua. This version of LuaEdit is the 2.2.1 release. It can be used, completly free of charges, for Personal and Commercial purposes and will always remain that way. You may suggest new features, report bugs or send any comments about LuaEdit via the Contact Us section of our Official Web Site.

This version of LuaEdit includes a built-in debugger for Lua 5.0 scripts, customized syntax highlighting, basic proposal library, basic parameter proposition, advanced and complete search engine, advanced and complete search and replace, bookmarking engine, advanced printing, project supports, quick tools, etc...

In the next release version of LuaEdit, the following features will be supported: remote debugging, header builder, Lua macros, Code profiling and any other suggested features...

What's new?

This version of LuaEdit includes major and minor bug fixes. LuaEdit is now faster and more efficient than ever. Also, some features have been upgraded for more conviviality when others are completely new to serve better the programmer's needs. Here is a detailed listing of the changes brought in this version:

- Major bug fix in the debug engine: A CPU Sleep instruction was called on every debug hook call wich was dramatically slowing down the code execution when running under LuaEdit.
- Major bug fix in the debug engine: When the unit is new, an extra call to ExpandUNCFileName was done wich was trowing an exeception.
- Minor bug fix in the tab control server: If prompted to save before closing a new unit, a save dialog was shown because the unit is new and an initial path is required to be specified by the user. Pressing the cancel button of that dialog was not really cancelling the action.
- Minor bug fix in the menus, submenus, popup menus and toolbars icons: The white pixels of the icons were treated and displayed as if they were transparent.
- Minor bug fix in the "Fucntion List" window: Sometimes this window wasn't automatically updating when switching from a unit to another.
- Changed the look of the status bar by handling a custom draw to make it look more like .Net style.
- Updated help files.
- Changed the "Project Tree" window into a new merge of a listview and a treeview into the same component
- Changed the "Watch" window into a new merge of a listview and a treeview into the same component. This now allow the user to better visualize variables, tables and sub-tables.
- New LuaEditDebug.dll file for embedded lua code debugging from user's applications. (Special thanks to Massimo Magnano)
- New library engine for completion proposal (not hardcoded anymore) including a new section in the editor settings: "Environment".
- New powerful "Find in Files" engine.
- New "Find Window 1" and "Find Window 2" windows to output the "Find in Files" results.

- New IE "Internal Browser" to allow browsing the internet inside the IDE.
- New "History" ring for the new "Internal Browser"
- New Comment/Uncomment, Uppercase/Lowercase selection commands
- New "Goto Last Edited" action to send the cursor back to the last edited line.
- Drag and drop *.lpr or *.lua files from Windows® into LuaEdit is now supported. This means that LuaEdit is opening the dropped files if they have the *.lpr or *.lua extension.

Minimum Requirements

LuaEdit is trying to be compatible with as many platform as possible. Officially, LuaEdit is Win32 compatible only. That means it is compatible on any Windows® platform. However, the following requirements should meet amply the needs for LuaEdit. If you ever have trouble running LuaEdit on any Win32 platform you may visit the Contact Us section of our Official Web Site to report your problem. Here is the minimum requirement list:

- 800MHz or faster
- Windows 98, 2000, XP
- 32MB RAM or more
- 25MB free hard disk space

LuaEdit

Copyright © 2004-2005 LuaEdit
This program is free software; you can redistribute it and/or modify it under the t
the Free Software Foundation; either version 2 of the License, or (at your option)

This program is distributed in the hope that it will be useful, but WITHOUT AN`
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

You should have received a copy of the GNU General Public License along with
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Lua 5.0

Copyright © 2003-2004 Tecgraf, PUC-Rio. Permission is hereby granted, free of
and associated documentation files (the "Software"), to deal in the Software with
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the
furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all co|
SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KINI
LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRI
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTI
OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHEI

GNU General Public License (GPL)

```
                                          Copyright (C
                                          59 Temple Pl

                                          Everyone is
```

of this lice

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.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. 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. 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.

2. 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. 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. 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. 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. 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. 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. 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. 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. 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. 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. 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

Mozilla Public License Version 1.1

1. Definitions.

1.0.1. "Commercial Use"
   means distribution or otherwise making the Covered Code available to a
   third party.
1.1. "Contributor"
   means each entity that creates or contributes to the creation of
   Modifications.
1.2. "Contributor Version"
   means the combination of the Original Code, prior Modifications used by a
   Contributor, and the Modifications made by that particular Contributor.
1.3. "Covered Code"
   means the Original Code or Modifications or the combination of the
   Original Code and Modifications, in each case including portions thereof.
1.4. "Electronic Distribution Mechanism"
   means a mechanism generally accepted in the software development
   community for the electronic transfer of data.
1.5. "Executable"
   means Covered Code in any form other than Source Code.
1.6. "Initial Developer"
   means the individual or entity identified as the Initial Developer in the
   Source Code notice required by Exhibit A.
1.7. "Larger Work"
   means a work which combines Covered Code or portions thereof with code
   not governed by the terms of this License.
1.8. "License"
   means this document.
1.8.1. "Licensable"
   means having the right to grant, to the maximum extent possible, whether
   at the time of the initial grant or subsequently acquired, any and all of the
   rights conveyed herein.
1.9. "Modifications"

   means any addition to or deletion from the substance or structure of either

the Original Code or any previous Modifications. When Covered Code is released as a series of files, a Modification is:

   a. Any addition to or deletion from the contents of a file containing Original Code or previous Modifications.
   b. Any new file that contains any part of the Original Code or previous Modifications.

1.10. "Original Code"
means Source Code of computer software code which is described in the Source Code notice required by Exhibit A as Original Code, and which, at the time of its release under this License is not already Covered Code governed by this License.

1.10.1. "Patent Claims"
means any patent claim(s), now owned or hereafter acquired, including without limitation, method, process, and apparatus claims, in any patent Licensable by grantor.

1.11. "Source Code"
means the preferred form of the Covered Code for making modifications to it, including all modules it contains, plus any associated interface definition files, scripts used to control compilation and installation of an Executable, or source code differential comparisons against either the Original Code or another well known, available Covered Code of the Contributor's choice. The Source Code can be in a compressed or archival form, provided the appropriate decompression or de-archiving software is widely available for no charge.

1.12. "You" (or "Your")
means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License or a future version of this License issued under Section 6.1. For legal entities, "You" includes any entity which controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty percent (50%) of the outstanding shares or beneficial ownership of such entity.

2. Source Code License.

2.1. The Initial Developer Grant.

The Initial Developer hereby grants You a world-wide, royalty-free, non-exclusive license, subject to third party intellectual property claims:

a. under intellectual property rights (other than patent or trademark) Licensable by Initial Developer to use, reproduce, modify, display, perform, sublicense and distribute the Original Code (or portions thereof) with or without Modifications, and/or as part of a Larger Work; and
b. under Patents Claims infringed by the making, using or selling of Original Code, to make, have made, use, practice, sell, and offer for sale, and/or otherwise dispose of the Original Code (or portions thereof).
c. the licenses granted in this Section 2.1 (a) and (b) are effective on the date Initial Developer first distributes Original Code under the terms of this License.
d. Notwithstanding Section 2.1 (b) above, no patent license is granted: 1) for code that You delete from the Original Code; 2) separate from the Original Code; or 3) for infringements caused by: i) the modification of the Original Code or ii) the combination of the Original Code with other software or devices.

2.2. Contributor Grant.

Subject to third party intellectual property claims, each Contributor hereby grants You a world-wide, royalty-free, non-exclusive license

a. under intellectual property rights (other than patent or trademark) Licensable by Contributor, to use, reproduce, modify, display, perform, sublicense and distribute the Modifications created by such Contributor (or portions thereof) either on an unmodified basis, with other Modifications, as Covered Code and/or as part of a Larger Work; and
b. under Patent Claims infringed by the making, using, or selling of Modifications made by that Contributor either alone and/or in combination with its Contributor Version (or portions of such combination), to make, use, sell, offer for sale, have made, and/or otherwise dispose of: 1)

Modifications made by that Contributor (or portions thereof); and 2) the combination of Modifications made by that Contributor with its Contributor Version (or portions of such combination).

c. the licenses granted in Sections 2.2 (a) and 2.2 (b) are effective on the date Contributor first makes Commercial Use of the Covered Code.

d. Notwithstanding Section 2.2 (b) above, no patent license is granted: 1) for any code that Contributor has deleted from the Contributor Version; 2) separate from the Contributor Version; 3) for infringements caused by: i) third party modifications of Contributor Version or ii) the combination of Modifications made by that Contributor with other software (except as part of the Contributor Version) or other devices; or 4) under Patent Claims infringed by Covered Code in the absence of Modifications made by that Contributor.

3. Distribution Obligations.

3.1. Application of License.

The Modifications which You create or to which You contribute are governed by the terms of this License, including without limitation Section 2.2. The Source Code version of Covered Code may be distributed only under the terms of this License or a future version of this License released under Section 6.1, and You must include a copy of this License with every copy of the Source Code You distribute. You may not offer or impose any terms on any Source Code version that alters or restricts the applicable version of this License or the recipients' rights hereunder. However, You may include an additional document offering the additional rights described in Section 3.5.

3.2. Availability of Source Code.

Any Modification which You create or to which You contribute must be made available in Source Code form under the terms of this License either on the same media as an Executable version or via an accepted Electronic Distribution Mechanism to anyone to whom you made an Executable version available; and if made available via Electronic Distribution Mechanism, must remain available for at least twelve (12) months after the date it initially became available, or at least six (6) months after a subsequent version of that particular Modification has been made available to such recipients. You are responsible for ensuring that the Source Code version remains available even if the Electronic Distribution Mechanism is maintained by a third party.

3.3. Description of Modifications.

You must cause all Covered Code to which You contribute to contain a file documenting the changes You made to create that Covered Code and the date of any change. You must include a prominent statement that the Modification is derived, directly or indirectly, from Original Code provided by the Initial Developer and including the name of the Initial Developer in (a) the Source Code, and (b) in any notice in an Executable version or related documentation in which You describe the origin or ownership of the Covered Code.

3.4. Intellectual Property Matters

(a) Third Party Claims

If Contributor has knowledge that a license under a third party's intellectual property rights is required to exercise the rights granted by such Contributor under Sections 2.1 or 2.2, Contributor must include a text file with the Source Code distribution titled "LEGAL" which describes the claim and the party making the claim in sufficient detail that a recipient will know whom to contact. If Contributor obtains such knowledge after the Modification is made available as described in Section 3.2, Contributor shall promptly modify the LEGAL file in all copies Contributor makes available thereafter and shall take other steps (such as notifying appropriate mailing lists or newsgroups) reasonably calculated to inform those who received the Covered Code that new knowledge has been obtained.

(b) Contributor APIs

If Contributor's Modifications include an application programming interface and Contributor has knowledge of patent licenses which are reasonably necessary to implement that API, Contributor must also include this information in the **legal** file.

(c) Representations.

Contributor represents that, except as disclosed pursuant to Section 3.4 (a) above, Contributor believes that Contributor's Modifications are Contributor's original creation(s) and/or Contributor has sufficient rights to grant the rights conveyed by this License.

3.5. Required Notices.

You must duplicate the notice in Exhibit A in each file of the Source Code. If it is not possible to put such notice in a particular Source Code file due to its structure, then You must include such notice in a location (such as a relevant directory) where a user would be likely to look for such a notice. If You created one or more Modification(s) You may add your name as a Contributor to the

notice described in [Exhibit A](). You must also duplicate this License in any documentation for the Source Code where You describe recipients' rights or ownership rights relating to Covered Code. You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Covered Code. However, You may do so only on Your own behalf, and not on behalf of the Initial Developer or any Contributor. You must make it absolutely clear than any such warranty, support, indemnity or liability obligation is offered by You alone, and You hereby agree to indemnify the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of warranty, support, indemnity or liability terms You offer.

3.6. Distribution of Executable Versions.

You may distribute Covered Code in Executable form only if the requirements of Sections [3.1](), [3.2](), [3.3](), [3.4]() and [3.5]() have been met for that Covered Code, and if You include a notice stating that the Source Code version of the Covered Code is available under the terms of this License, including a description of how and where You have fulfilled the obligations of Section [3.2](). The notice must be conspicuously included in any notice in an Executable version, related documentation or collateral in which You describe recipients' rights relating to the Covered Code. You may distribute the Executable version of Covered Code or ownership rights under a license of Your choice, which may contain terms different from this License, provided that You are in compliance with the terms of this License and that the license for the Executable version does not attempt to limit or alter the recipient's rights in the Source Code version from the rights set forth in this License. If You distribute the Executable version under a different license You must make it absolutely clear that any terms which differ from this License are offered by You alone, not by the Initial Developer or any Contributor. You hereby agree to indemnify the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of any such terms You offer.

3.7. Larger Works.

You may create a Larger Work by combining Covered Code with other code not governed by the terms of this License and distribute the Larger Work as a single product. In such a case, You must make sure the requirements of this License

are fulfilled for the Covered Code.

4. Inability to Comply Due to Statute or Regulation.

If it is impossible for You to comply with any of the terms of this License with respect to some or all of the Covered Code due to statute, judicial order, or regulation then You must: (a) comply with the terms of this License to the maximum extent possible; and (b) describe the limitations and the code they affect. Such description must be included in the **legal** file described in Section 3.4 and must be included with all distributions of the Source Code. Except to the extent prohibited by statute or regulation, such description must be sufficiently detailed for a recipient of ordinary skill to be able to understand it.

5. Application of this License.

This License applies to code to which the Initial Developer has attached the notice in Exhibit A and to related Covered Code.

6. Versions of the License.

6.1. New Versions

Netscape Communications Corporation ("Netscape") may publish revised and/or new versions of the License from time to time. Each version will be given a distinguishing version number.

6.2. Effect of New Versions

Once Covered Code has been published under a particular version of the License, You may always continue to use it under the terms of that version. You may also choose to use such Covered Code under the terms of any subsequent version of the License published by Netscape. No one other than Netscape has the right to modify the terms applicable to Covered Code created under this License.

6.3. Derivative Works

If You create or use a modified version of this License (which you may only do in order to apply it to code which is not already Covered Code governed by this License), You must (a) rename Your license so that the phrases "Mozilla", "MOZILLAPL", "MOZPL", "Netscape", "MPL", "NPL" or any confusingly similar phrase do not appear in your license (except to note that your license differs from this License) and (b) otherwise make it clear that Your version of the license contains terms which differ from the Mozilla Public License and Netscape Public License. (Filling in the name of the Initial Developer, Original Code or Contributor in the notice described in Exhibit A shall not of themselves be deemed to be modifications of this License.)

7. Disclaimer of warranty

Covered code is provided under this license on an "as is" basis, without warranty of any kind, either expressed or implied, including, without limitation, warranties that the covered code is free of defects, merchantable, fit for a particular purpose or non-infringing. The entire risk as to the quality and performance of the covered code is with you. Should any covered code prove defective in any respect, you (not the initial developer or any other contributor) assume the cost of any necessary servicing, repair or correction. This disclaimer of warranty constitutes an essential part of this license. No use of any covered code is authorized hereunder except under this disclaimer.

8. Termination

8.1. This License and the rights granted hereunder will terminate automatically if You fail to comply with terms herein and fail to cure such breach within 30 days of becoming aware of the breach. All sublicenses to the Covered Code which are properly granted shall survive any termination of this License. Provisions which, by their nature, must remain in effect beyond the termination of this License shall survive.

8.2. If You initiate litigation by asserting a patent infringement claim (excluding declatory judgment actions) against Initial Developer or a Contributor (the Initial Developer or Contributor against whom You file such action is referred to as "Participant") alleging that:

a. such Participant's Contributor Version directly or indirectly infringes any patent, then any and all rights granted by such Participant to You under Sections 2.1 and/or 2.2 of this License shall, upon 60 days notice from Participant terminate prospectively, unless if within 60 days after receipt of notice You either: (i) agree in writing to pay Participant a mutually agreeable reasonable royalty for Your past and future use of Modifications made by such Participant, or (ii) withdraw Your litigation claim with respect to the Contributor Version against such Participant. If within 60 days of notice, a reasonable royalty and payment arrangement are not mutually agreed upon in writing by the parties or the litigation claim is not withdrawn, the rights granted by Participant to You under Sections 2.1 and/or 2.2 automatically terminate at the expiration of the 60 day notice period specified above.
b. any software, hardware, or device, other than such Participant's Contributor Version, directly or indirectly infringes any patent, then any rights granted to You by such Participant under Sections 2.1(b) and 2.2(b) are revoked effective as of the date You first made, used, sold, distributed, or had made, Modifications made by that Participant.

8.3. If You assert a patent infringement claim against Participant alleging that such Participant's Contributor Version directly or indirectly infringes any patent where such claim is resolved (such as by license or settlement) prior to the initiation of patent infringement litigation, then the reasonable value of the

licenses granted by such Participant under Sections 2.1 or 2.2 shall be taken into account in determining the amount or value of any payment or license.

8.4. In the event of termination under Sections 8.1 or 8.2 above, all end user license agreements (excluding distributors and resellers) which have been validly granted by You or any distributor hereunder prior to termination shall survive termination.

9. Limitation of liability

Under no circumstances and under no legal theory, whether tort (including negligence), contract, or otherwise, shall you, the initial developer, any other contributor, or any distributor of covered code, or any supplier of any of such parties, be liable to any person for any indirect, special, incidental, or consequential damages of any character including, without limitation, damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses, even if such party shall have been informed of the possibility of such damages. This limitation of liability shall not apply to liability for death or personal injury resulting from such party's negligence to the extent applicable law prohibits such limitation. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so this exclusion and limitation may not apply to you.

10. U.S. government end users

The Covered Code is a "commercial item," as that term is defined in 48 C.F.R. 2.101 (Oct. 1995), consisting of "commercial computer software" and "commercial computer software documentation," as such terms are used in 48 C.F.R. 12.212 (Sept. 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (June 1995), all U.S. Government End Users acquire Covered Code with only those rights set forth herein.

11. Miscellaneous

This License represents the complete agreement concerning subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. This License shall be governed by California law provisions (except to the extent applicable law, if any, provides otherwise), excluding its conflict-of-law provisions. With respect to disputes in which at least one party is a citizen of, or an entity chartered or registered to do business in the United States of America, any litigation relating to this License shall be subject to the jurisdiction of the Federal Courts of the Northern District of California, with venue lying in Santa Clara County, California, with the losing party responsible for costs, including without limitation, court costs and reasonable attorneys' fees and expenses. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any law or regulation which provides that the language of a contract shall be construed against the drafter shall not apply to this License.

12. Responsibility for claims

As between Initial Developer and the Contributors, each party is responsible for claims and damages arising, directly or indirectly, out of its utilization of rights under this License and You agree to work with Initial Developer and Contributors to distribute such responsibility on an equitable basis. Nothing herein is intended or shall be deemed to constitute any admission of liability.

13. Multiple-licensed code

Initial Developer may designate portions of the Covered Code as "Multiple-Licensed". "Multiple-Licensed" means that the Initial Developer permits you to utilize portions of the Covered Code under Your choice of the MPL or the alternative licenses, if any, specified by the Initial Developer in the file described in Exhibit A.

Exhibit A - Mozilla Public License.

"The contents of this file are subject to the Mozilla Public License
                                        Version 1.1
                                        compliance w
                                        http://www.m

                                        Software dis
                                        basis, WITHO
                                        License for
                                        under the Li

                                        The Initial
                                        Portions cre
                                        All Rights F

                                        Contributor(

LuaEdit is a script editor designed to simplify the development of projects created with Lua. Using LuaEdit you can create scripts faster than before and finally be able to trace it in runtime.

LuaEdit provides a suite Rapid Development Tools (RDT), including completion proposal, parameters proposition, syntax highlights and more. Also, you will find in LuaEdit that convivial environment found in the famous Visual Studio .Net$^{®}$ with auto-hide dockable windows.

This chapter brievly describes the LuaEdit environment with examples and colorful images.

Once you started LuaEdit, the Integrated Development Invironment (also called the IDE) is immediately presented to you. This IDE provides you tools to develop and debug Lua scripts with a shorter development time.

When developping with the IDE, you will notice that there is mainly two important environment:

- The designtime environment
- The runtime environment

The designtime environment will cover important actions and tools such as copy, cut, paste, search, completion proposal list, breakpoint managing window, etc. The runtime environment will cover more tools and actions such as step over, step into, local and global variables lists, call stack and lua stack windows, etc. Designtime tools and actions will usually still be available in the runtime environment (also called debugging) when the runtime environment tools and actions won't be actives in the designtime environment. (also called programming)

Working with Projects

Projects in LuaEdit can be very useful in many ways. They can group and manage files better than single units without projects. They also feature debug and revision informations.
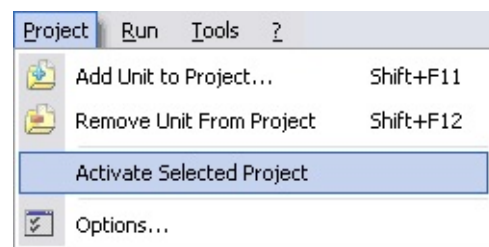

Picture 1.1

The content of a project may consulted through the "Project Tree" window (Picture 1.1). In this window, single units and projects (including their member units) are displayed in a hierarchical view:
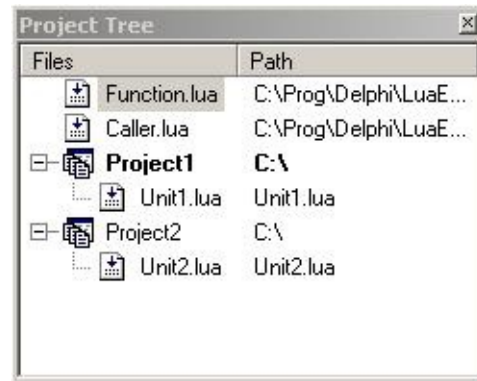
```
+ --- Single Units
¦
¦
+ --- Project
   ¦
   ¦
    + --- Units
```

Because more than one project at once can be opened, the idea of "Active Project" is used in synchronization with the "Project" menu (Picture 1.2). This means that all modifications made through this menu will apply to the current "Active Project". In order to know if a project is active or not, have a look at the "Project Tree" window

Picture 1.2

wich will display the "Active Project" in bold. Those actions are also available by right-clicking on the "Active Project" node. To remove or add units to the "Active Project" simply click the "Project/Add Unit to Project" menu or the "Project/Remove Unit From Project" menu.

You can edit a unit by double-clicking on a unit node in the "Project Tree". This will add the unit in the edition tab and will display the code.

To display or hide the "Project Tree", simply use the menu "View/Project Tree".

Many basic actions and features are available in LuaEdit. The shortcuts and icons of those features and actions have been adapted so that they can looks and reacts the same way as they do in Visual Studio .Net [®]. This section will briefly describe them.

File handling

LuaEdit recognize two type of file: *.lpr extension standing for "Lua Project" and *.lua extension standing for "Lua Units". To open any *.lpr files, select the menu "File/Open Lua Project..." or the menu "File/Open File...". To open any *.lua files, select the menu "File/Open File...". The selected file will be added to the "Project Tree Window". The menu "File/Save" will save the currently edited unit on its originally opened localtion. The menu "File/Save As..." will display a save dialog wich allow to save the file anywhere you can. The menu "File/Save All" will save any modified or new units and projects. The menu "File/Save Project As..." will display a save dialog so that you can save the current "Active Project" anywhere you can. New units and projects can be created via the menu "File/New/Unit" and "File/New/Project". The menu "File/Reopen" contains all recently opened projects and units. Simply click on any of these to reopen it. The menu "File/Print Steup..." will display the printing options and the menu "File/Print..." will display a print dialog to print the currently edited unit.

Code handling

Luaedit exposed all basic code handling actions. The menu "Edit/Undo" will undo the lastest changes and the menu "Edit/Redo" will redo the lastest *undoed* changes. The menu "Edit/Cut" will cut the current selection from the currently edited unit and will add this text in the clipboard. The menu "Edit/Copy" will copy the current selection from the currently edited unit and will add this text in the clipboard. The menu "Edit/Paste" will insert the text in the clipboard at the currrent cursor position of the currently edited unit. The "Edit/Select All" menu will select all the text present in the currently edited unit. The "Edit/Find..." menu will display a find dialog so that you can easily find the text you are looking for. The menu "Edit/Replace..." will display a find and replace dialog so that you can easily replace the text you are looking for.

The "Edit/Find Again" menu will continue the previous search keeping the same given parameters. The menu "Edit/Find in Files" will display a find in files dialog so that you can find text contained in some files of your hard drive with ease (NOTE: the find in files engine only find text in *.lua files if any). The menu "Edit/Indent Selection" will increase the current selection of the currently edited unit of one level according to your tab settings. The menu "Edit/Unindent Selection" will decrease the current selection of the currently edited unit of one level according to your tab settings. The "Edit/Comment Selection" and the "Edit/Uncomment Selection" menus will comment/uncomment the selected text in one click. The "Edit/Uppercase Selection" and the "Edit/Lowercase Selection" menus will set all characters of the current selection to upper or lower case. The menu "Edit/Goto Last Edited" will bring you to the last edited line. The "Edit/Go to Line..." menu will display a dialog to enter the line number where to go in the currently edited unit.

Just like other IDEs, LuaEdit allows customization of its own environment. This section will explain the different modifications available through the "Editor Settings Window". This window, available via the menu "Tools/Editor Settings...", is divisied in three major categories: "General", "Display" and "Colors".

The General Tab (Picture 1.1)
This tab featrues miscellaneous options for the code editor and the IDE.
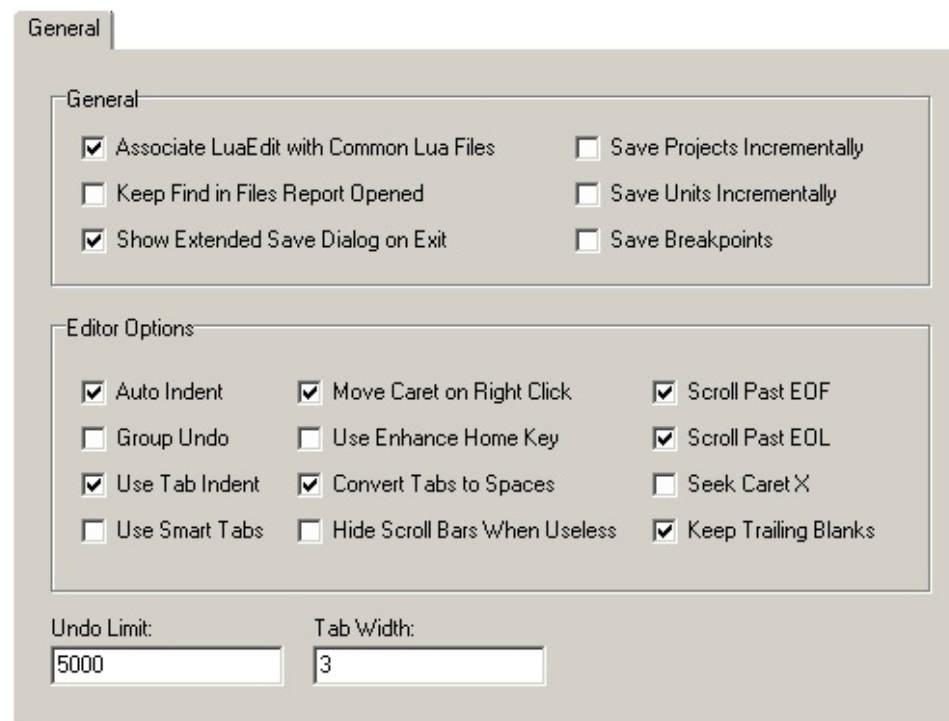
Picture 1.1

- **"Associate LuaEdit with Common Lua File"**
  Check this option to associate the common *.lua and *.lpr files with LuaEdit in Windows. When checked, LuaEdit will ask for a system reboot in order for those changes to take effect. Once done, it is now possible to simply double-click on *.lua or *.lpr files and LuaEdit will handle automatic opening.

- **"Keep Find in Files Report Opened"**
Check this option to make sure that the find in files report will stay open once the search is done. This can be useful to actually read the final statistics of the search.
- **"Save Breakpoints"**
Check this option in order to allow LuaEdit saving the breakpoints information such as their conditions and positions while saving a file. Next time you will reopen this file, the breakpoints should be reinitialized.
- **"Show Extended Save Dialog on Exit"**
Check this option in order to display a window wich groups all new or modified files opened that might need to be saved when exiting LuaEdit. Without this option, LuaEdit will popup a message box for every file that required a save on after the other.
- **"Save Projects Incrementally"**
Check this option in order to save *.lpr files in an incremental way. This means that if this option is checked and you opened a file called "Project5.lpr", it will automatically save the file as "Project6.lpr" next time you will save it. Just like "Project.lpr" will result into "Project1.lpr". This can be a good, safe and quick way to backup files everytime you save it. (3D studio max has the same kind of feature)
- **"Save Units Incrementally"**
Check this option in order to save *.lua files in an incremental way. This means that if this option is checked and you opened a file called "Unit9.lua", it will automatically save the file as "Unit10.lpr" next time you will save it. Just like "Unit.lpr" will result into "Unit1.lpr". This can be a good, safe and quick way to backup files everytime you save it. (3D studio max has the same kind of feature)
- **"Auto Indent"**
This option will make the code editor indenting the caret on new lines with the same amount of leading white spaces or tabs as the preceding line.
- **"Group Undo"**
This will make undo/redo actions to handle all continous changes of the same kind in one call instead undoing/redoing each changes separately on after the other.
- **"Use Tab Indent"**
Check this option in order to simulate the indent/unindent actions on the current selection by pressing the [Tab] and [Shift+Tab].
- **"Use Smart Tabs"**
Check this option in order to send the cursor to the next non-white space

character of the previous line when pressing the [Tab] key.

- **"Move Caret on Right Click"**
  Check this option in order to move the caret position when right clicking on the code editor. (This will still popup the code editor popup menu)
- **"Use Enhance Home Key"**
  Check this option in order to send the cursor to the first non-white space character of the current line instead of the very begining of the current line. (If checked, press twice to do so)
- **"Convert Tabs to Spaces"**
  Check this option in order to converts tab characters into the specified number of space characters for a tab.
- **"Hide Scroll Bars When Useless"**
  Check this option in order to hide the scroll bars whenever they are disabled because they can't scroll.
- **"Scroll Past EOF"**
  Check this option in order to allow scrolling even though the End Of File is reached.
- **"Scroll Past EOL"**
  Check this option in order to allow scrolling even though the End Of Line is reached.
- **"Save Breakpoints"**
  Check this option in order to keep the X position of the caret when moving through lines (Takes effects only if "Scoll Past EOL" is NOT check)
- **"Keep Trailing Blanks"**
  If this option is checked, spaces at the End Of Lines will be trimmed and not saved.
- **"Undo Limits"**
  Set in this field the number of undo allowed to LuaEdit. The higher this will be, the more memory LuaEdit might take for Undo/Redo actions only.
- **"Tab Width"**
  Set in this field the width of a tab character in spaces. For example, if this setting is set to 4, a tab character will have the equivalent of 4 white spaces.

The Environement Tab (Picture 1.2)
This tab features some environement features such as search paths.
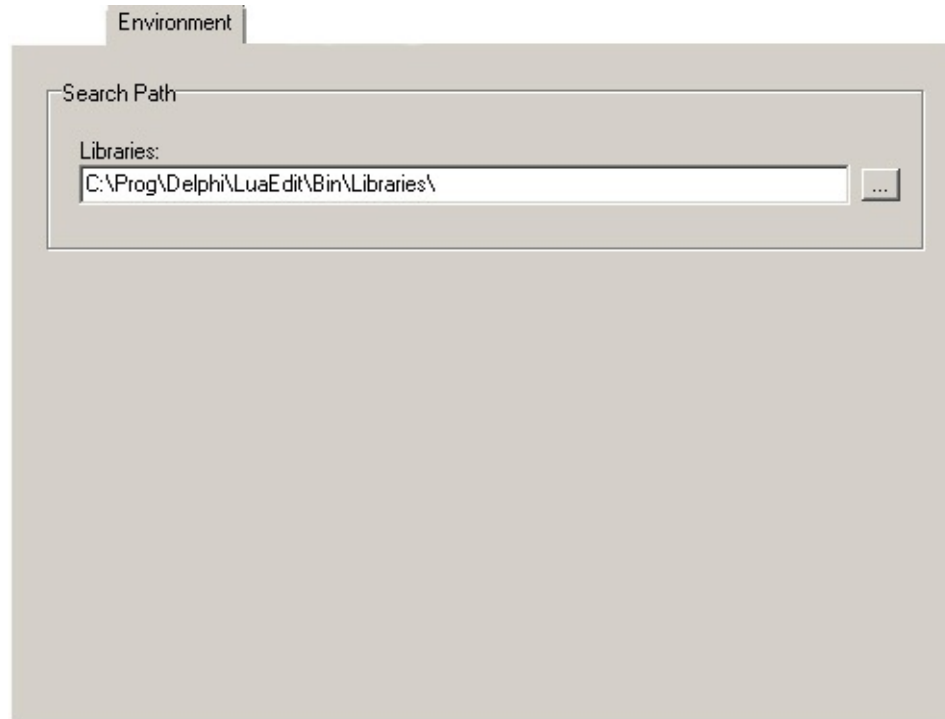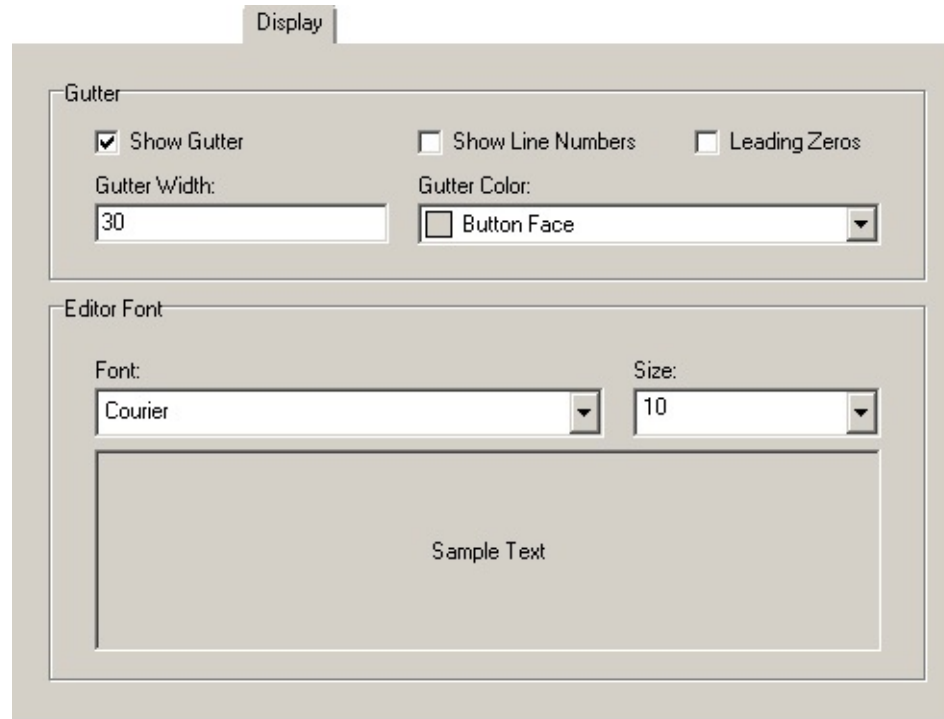
Picture 1.2

- **"Libraries"**
  This option is a string separated by commas to indicate all the search paths for the libraries. Libraries are *.lib files containing function definition datas for the completion proposal engine to use. You can in any time create your own *.lib file following the same format as the basic Lua50.lib and LuaEdit should consider it and retreive the data if the path where this file is located is specified in this option. To make things easier than just writing the paths yourself, a search path manager window is available by pressing the "..." button next-to the text box.

The Display Tab (Picture 1.3)
This tab features display options for the gutter and the code editor.

- **"Show Gutter"**

Picture 1.3

 Check/uncheck this option in order to display/hide the gutter on the code editor.
- **"Show Line Numbers"**
 Check/uncheck this option in order to display/hide the line numbers in the gutter.
- **"Leading Zeros"**
 Check/uncheck this option in order to display/hide leading zeros for the line numbers in the gutter. (Works only if the "Show Line Numbers" option is check)
- **"Gutter Width"**
 Set in this field the total width of the gutter in pixels. (excluding the line numbers because they make the gutter's width beeing recalculated every time they display or hide)
- **"Gutter Color"**

Choose the color you want for the gutter to be painted with. (The colors proposed in the list and the colors you can choose on the light spectrum may vary according to your Operating System and its Display Settings)
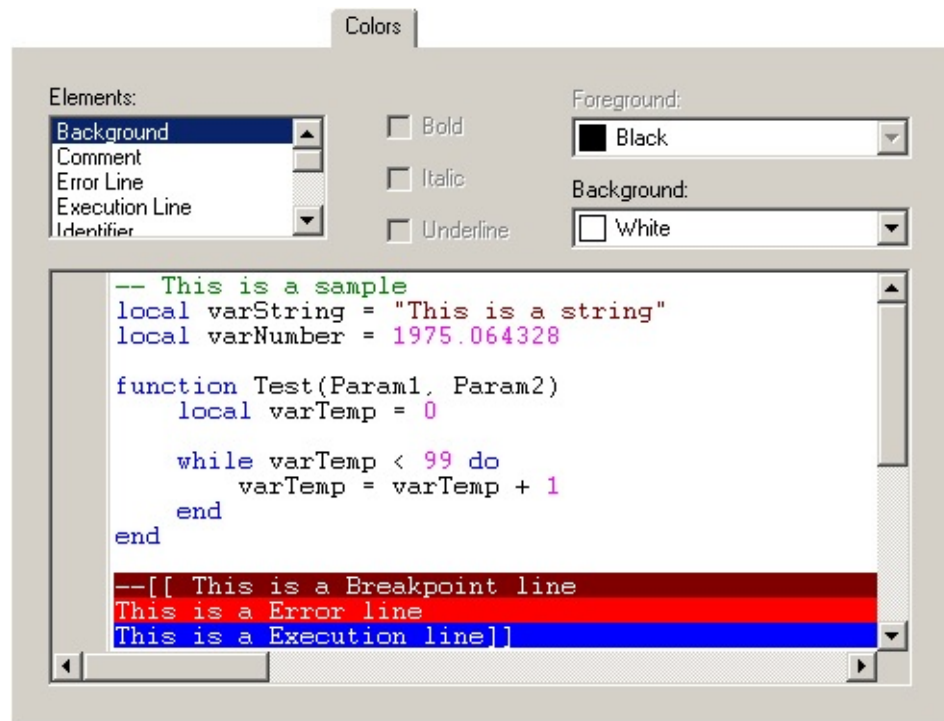- **"Font"**
Choose the type of font that will be used to display the code in the code editor. (The fonts proposed in the list may vary according to your Operating System)
- **"Size"**
Set in this field the size of the font in points to be used to display the code in the code editor.

The Colors Tab (Picture 1.4)
This tab features color options for the the code editor only.

Picture 1.4

- **"Elements"**
The item selected in this list will be afected by all other options in this tab. This list groups all kind of tokens in Lua scripts such as strings and numbers.

- **"Bold"**

  Check this option in order to make the selected token being bold in the code editor.
- **"Italic"**

  Check this option in order to make the selected token being italic in the code editor.
- **"Underline"**

  Check this option in order to make the selected token being underline in the code editor.
- **"Foreground"**

  Choose the color you want for the selected token's foreground to be painted with. (The colors proposed in the list and the colors you can choose on the light spectrum may vary according to your Operating System and its Display Settings)
- **"Background"**

  Choose the color you want for the selected token's background to be painted with. (The colors proposed in the list and the colors you can choose on the light spectrum may vary according to your Operating System and its Display Settings)

The LuaEdit code editor acts and proposes features seen in other famous code editors such as Visual C++®, Delphi®, etc. It features syntax highlighting, completion proposal list, bookmarking and more. This section will summarize and briefly explain those features.

- **Syntax Highlighting**

  When programming with the LuaEdit code editor, you will notice that keywords such as "if", "function" and "while" are automatically highlighted according to your highlitghting settings. Strings, comments and numbers are also automatically highlighted. For strings and comments, they can be in some circumstaces multiline highlighted. For example, all code, spread on several lines or not, included in --[[and --]] delimiters are highlighted as comments because it is also a Lua 5.0 feature. Same thing for strings with [[and]] delimiters.
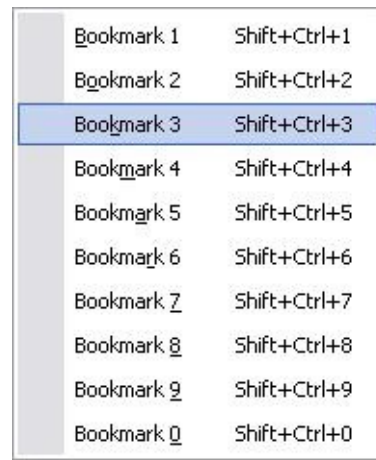
- **Completion Proposal List**

  LuaEdit also provides some integrated tools to shorten development time. The completion proposal list is a list that displays all matching function, libraries, global variables, etc with the text on left of the cursor. To popup this list while developping press Ctrl+Space (For physical and logical reasons, the local and global variables are not shown in the list unless they are specified in a *.lib file in one of the specified search paths (See the Editor Settings section for more details)). Once displayed, you may choose an item of the proposed list by double-clicking on that item or pressing one of the following keys: "Enter", "Space", ".", ")" or "(". To hide the list, simply ckick anywhere else than on the list or press "Escape".

- **Parameter Proposition**

When writing the call of a function you may want to use the parameter proposition tool. This tool display while programming the parameters to enter for the that function call. To display the parameter proposition tools press Ctrl+Shift+Space. The parameter in bold is the current paramter to enter.

- **Bookmarking**

  Bookmarks have been designed to easily and quickly switch from a place in a script to another. This can save lots of development time and prevent confusions while developing. Up to 10 bookmarks per unit are available to use anywhere in a script. To toggle a bookmark press Ctrl+Shift+1 for bookmark #1 or Ctrl+Shift+2 for bookmark #2 and so on or use the code editor popup menu by right-clicking on the code editor and finally "Toggle Bookmark" (Picture 1.1). To go to a toggled bookmark press Shift+1 for Bookmark #1 or Shift+2 for Bookmark #2 and so on or use the code editor popup menu by right-clicking on the code editor and finally "Goto Bookmark" (Similar to Picture 1.1)

Bookmark 1	Shift+Ctrl+1
Bookmark 2	Shift+Ctrl+2
Bookmark 3	Shift+Ctrl+3
Bookmark 4	Shift+Ctrl+4
Bookmark 5	Shift+Ctrl+5
Bookmark 6	Shift+Ctrl+6
Bookmark 7	Shift+Ctrl+7
Bookmark 8	Shift+Ctrl+8
Bookmark 9	Shift+Ctrl+9
Bookmark 0	Shift+Ctrl+0

  Picture 1.1

- **Regular Actions**

  Of course, the code editor exposed regular actions such as cut, copy, paste, block indent, block unindent, etc. All those actions are available through the "Edit" menu in LuaEdit and some of them are also available in the code editor popup menu.

Debugging is long and difficult task to do. It is also a crucial state in programming project because this task have to be well done in order

Picture 1.1

to bring a project successful. LuaEdit provides servaral tools to get this *painful* task done as easy as possible. This section will explain the basic actions to debug scripts with LuaEdit.

Basics

First of all, you must press the "Run" (F5) button on the "Debug Toolbar" (Picture 1.1) or by using the menu "Run/Run Script" in order to get into a debug session. This will load the script into a "Lua State" (See Lua 5.0 documentation available through the menu "?/Lua Help") and execute each valid line of you script until it reaches the end or a valid breakpoint (See documentation below for more informations about breakpoints). If an error happened while running the script, LuaEdit will display the message through a message box and through the "Messages Window" (Picture 1.2). Several debugging informations are available, once the script execution breaks on a line. Local and Global variables values are available through the "Local Variables Window" and the "Global Variables Window". Also, the "Lua Stack Window" shows all Lua's stacked variables values at the moment of the current execution and the "Call Stack Window" shows the history of all function calls made from the current script. This call stack may be useful to trace back the values of the local and global variables in previous calls.

Step by Step

A few choices are available to pursue the execution of the script. The "Step Over" (F10) and "Step Into" (F11) buttons allows to jump the function call of the current debugging line if any. If "Step into"
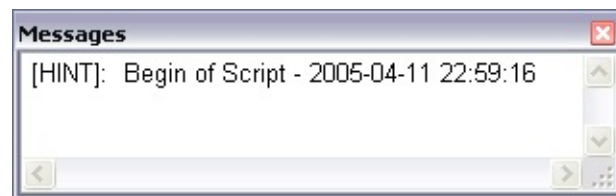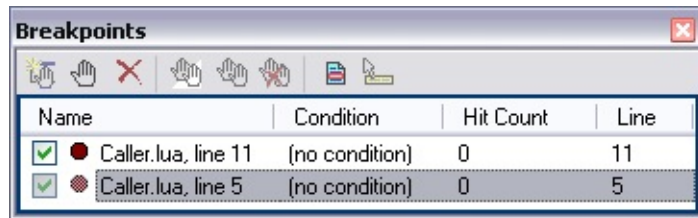

Picture 1.2

is pressed and no function call is made on the current debugging line, the debugger will simulate the "Step Over" feature wich simply jump to the next

executed line. The "Run Script to Cursor" (Ctrl+F10) button will run the script until it reaches the line where cursor is currently positioned or until the end of the script. The "Run" (F5) button will pursue the execution of the scipt if already started.

Manipulating Breakpoints

Breakpoins are very useful when debugging. To insert a breakpoint at the line where the cursor is, press the "Toggle Breakpoint" (F9) button from the debug toolbar (Picture 1.2). When debugging a script, the execution should break on a line where a breakpoint has been detected. Advanced manipulation and features for breakpoint are available through the "Breakpoints Window" (Picture 1.3) such as disabling/enabling breakpoints.
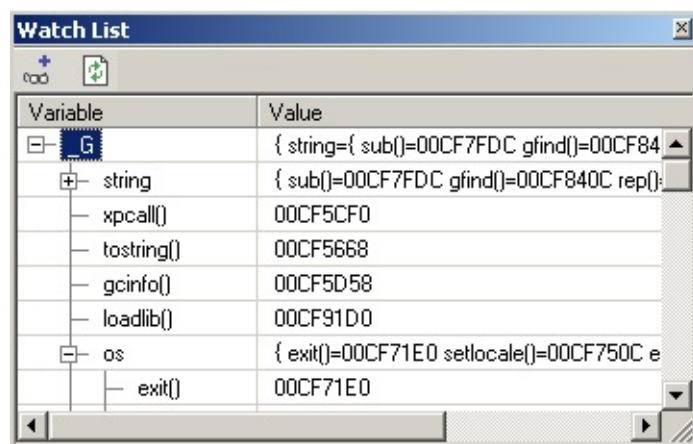

Picture 1.3

A disabled breakpoint won't break the execution of the script but will leave the breakpoint marker (maroon/grayed) on the code editor. To disable/enable a breakpoint, simply check/uncheck the checkbox on the left of the breakpoint name in the "Breakpoints Window". You can also set a condition on a breakpoint wich will be evaluated when the scipt is runnning to determine whether or not it will break on this breakpoint. The evaluation of the condition should always return "true" or "false". You can also remove, add and goto breakpoints through the "Breakpoints Window".

Watching Variables

In Lua, just like other languages, variables are very important for the code to be working. Variables in programming are abstract concept since they are located in the RAM and that their content may change several times in a few operations. That's why the "Watch" window (Picture 1.4) becomes

Picture 1.4

a useful tool for the programmer. Since Lua logic is based on tables, LuaEdit has adapted a watch window wich act like the MS Visual Studio .Net® one. This

means that all the content of any tables (variables and nested tables) can be explored in a hierarchic way. This can be extremely useful when complex tables are part of your code. The watch window also display regular variables content.

NOTE: Any complex data such as user data and functions will be displayed as a pointer address. See picture 1.4 for details.

Using the Initializer

LuaEdit offer the possibility to bind the code to a Dynamic-Link Library (DLL) before the execution of the script. This can be very useful if any function call in the scripts refer to some C++ functions defined in such a file. By using the initializer, you will prevent error raised by Lua 5.0 because it can't find the reference of the function call. To set the initializer, just popup the "Project Options Window" by clicking on the "Project/Options..." menu. Once the windowis displayed, select the "Debug" tab and enter the full path of the DLL or simply click on the browse button and open the DLL from the browse dialog. In order to make this feature working your DLL must export a function of one of these prototype according to the right language it has been built from:

- Delphi: **LuaDebug_Initializer(L: PLua_State): Integer; cdecl;**
- C/C++: **int LuaDebug_Initializer(lua_State *L);**

NOTE: The initializer property is only available when the unit is part of a project.

Using LuaEditDebug.dll

LuaEdit has been compiled and adapted into a Dynamic-Link Library (DLL) file. This now allowing embedded lua code in executable files (EXE) to be debug in LuaEdit. First of all, in order to use it, the lua state structure must be opened AND closed by the dll. To do so, here are the two function prototypes to declare in your application code:

- Delphi:
    - **function LuaEditDebugOpen: Plua_State;** // **Function to open the lua_state structure with LuaEdit**
    - **procedure LuaEditDebugClose(LuaState: Plua_State);** // **Function to close the lua_state structure with LuaEdit**
- C/C++:
    - **lua_State *LuaEditDebugOpen(void);** // **Function to open the**

lua_state structure with LuaEdit

- **void LuaEditDebugClose(lua_State *LuaState);   // Function to close the lua_state structure with LuaEdit**

By calling the LuaEditDebugOpen function, LuaEdit will perform a call to the lua_open Lua API function and return the state as a pointer. By calling the LuaEditDebugClose function, LuaEdit will perform a call to the lua_close Lua API function. Make sure that only LuaEdit is opening AND closing the state structure. If you intend to use LuaEdit for this purpose, your application should never open or close the state. To begin the debug session in LuaEdit, two way are available. The first one is by specifying a file name and the second one is by giving the lua code. Once called, both functions will immediately start debuging the code in LuaEdit. Here are their prototypes:

- Delphi:
  - **function LuaEditDebugStart(LuaState: Plua_State; Code: PChar): Integer;**
  - **function LuaEditDebugStartFile(LState: Plua_State; Filename: PChar): Integer;**
- C/C++:
  - **int LuaEditDebugStart(lua_State *LuaState, char *Code);**
  - **int LuaEditDebugStartFile(lua_State *LState, char *Filename);**

Reference manual for Lua 5.0

Lua 5.0 Reference Manual [top | index | ps | pdf]

Last update: Wed May 7 18:34:34 EST 2003

Lua 5.0 Reference Manual

by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes

1 - Introduction

Lua is an extension programming language designed to support general procedural programming with data description facilities. It also offers good support for object-oriented programming, functional programming, and data-driven programming. Lua is intended to be used as a powerful, light-weight configuration language for any program that needs one. Lua is implemented as a library, written in *clean* C (that is, in the common subset of ANSI C and C++).

Being an extension language, Lua has no notion of a "main" program: it only works *embedded* in a host client, called the *embedding program* or simply the *host*. This host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code. Through the use of C functions, Lua can be augmented to cope with a wide range of different domains, thus creating customized programming languages sharing a syntactical framework.

The Lua distribution includes a stand-alone embedding program, `lua`, that uses the Lua library to offer a complete Lua interpreter.

Lua is free software, and is provided as usual with no guarantees, as stated in its copyright notice. The implementation described in this manual is available at Lua's official web site, `www.lua.org`.

Like any other reference manual, this document is dry in places. For a discussion of the decisions behind the design of Lua, see the papers below, which are available at Lua's web site.

- R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua---an extensible extension language. *Software: Practice & Experience* **26** #6 (1996) 635-652.
- L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. The design and implementation of a language for extending applications. *Proceedings of XXI Brazilian Seminar on Software and Hardware*

(1994) 273-283.

- L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. Lua: an extensible embedded language. *Dr. Dobb's Journal* **21** #12 (Dec 1996) 26-33.
- R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of an extension language: a history of Lua, *Proceedings of V Brazilian Symposium on Programming Languages* (2001) B-14-B-28.

Lua means "moon" in Portuguese and is pronounced LOO-ah.

2 - The Language

This section describes the lexis, the syntax, and the semantics of Lua. In other words, this section describes which tokens are valid, how they can be combined, and what their combinations mean.

The language constructs will be explained using the usual extended BNF, in which {*a*} means 0 or more *a*'s, and [*a*] means an optional *a*. Non-terminals are shown in *italics*, keywords are shown in **bold**, and other terminal symbols are shown in `typewriter` font, enclosed in single quotes.

2.1 - Lexical Conventions

Identifiers in Lua can be any string of letters, digits, and underscores, not beginning with a digit. This coincides with the definition of identifiers in most languages. (The definition of letter depends on the current locale: any character considered alphabetic by the current locale can be used in an identifier.)

The following *keywords* are reserved and cannot be used as identifiers:

```
and       break    do      else      elseif
end       false    for     function  if
in        local    nil     not       or
repeat    return   then    true      until      while
```

Lua is a case-sensitive language: `and` is a reserved word, but `And` and `AND` are two different, valid identifiers. As a convention, identifiers starting with an underscore followed by uppercase letters (such as `_VERSION`) are reserved for internal variables used by Lua.

The following strings denote other tokens:

```
+     -      *      /      ^      =
~=    <=     >=     <      >      ==
(     )      {      }      [      ]
;     :      ,      .      ..     ...
```

Literal strings can be delimited by matching single or double quotes, and can contain the following C-like escape sequences:

- `\a` --- bell
- `\b` --- backspace
- `\f` --- form feed
- `\n` --- newline
- `\r` --- carriage return
- `\t` --- horizontal tab
- `\v` --- vertical tab
- `\\` --- backslash

- `\"` --- quotation mark
- `\'` --- apostrophe
- `\[` --- left square bracket
- `\]` --- right square bracket

Moreover, a `\newline´ (that is, a backslash followed by a real newline) results in a newline in the string. A character in a string may also be specified by its numerical value using the escape sequence `\ddd´, where *ddd* is a sequence of up to three decimal digits. Strings in Lua may contain any 8-bit value, including embedded zeros, which can be specified as `\0´.

Literal strings can also be delimited by matching double square brackets `[[ · · · ]]`. Literals in this bracketed form may run for several lines, may contain nested `[[ · · · ]]` pairs, and do not interpret any escape sequences. For convenience, when the opening `[[´ is immediately followed by a newline, the newline is not included in the string. As an example, in a system using ASCII (in which `a´ is coded as 97, newline is coded as 10, and `1´ is coded as 49), the four literals below denote the same string:

```
(1)    "alo\n123\""
(2)    '\97lo\10\04923"'
(3)    [[alo
       123"]]
(4)    [[
       alo
       123"]]
```

Numerical constants may be written with an optional decimal part and an optional decimal exponent. Examples of valid numerical constants are

```
   3     3.0     3.1416  314.16e-2   0.31416E1
```

Comments start anywhere outside a string with a double hyphen (`--`). If the text immediately after `--` is different from `[[`, the comment is a *short comment*, which runs until the end of the line. Otherwise, it is a *long comment*, which runs until the corresponding `]]`. Long comments may run for several lines and may contain nested `[[ · · · ]]` pairs.

For convenience, the first line of a chunk is skipped if it starts with #. This facility allows the use of Lua as a script interpreter in Unix systems (see [6](#)).

2.2 - Values and Types

Lua is a *dynamically typed language*. That means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

There are eight basic types in Lua: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread*, and *table*. *Nil* is the type of the value **nil**, whose main property is to be different from any other value; usually it represents the absence of a useful value. *Boolean* is the type of the values **false** and **true**. In Lua, both **nil** and **false** make a condition false; any other value makes it true. *Number* represents real (double-precision floating-point) numbers. (It is easy to build Lua interpreters that use other internal representations for numbers, such as single-precision float or long integers.) *String* represents arrays of characters. Lua is 8-bit clean: Strings may contain any 8-bit character, including embedded zeros (`'\0'`) (see [2.1](#)).

Functions are *first-class values* in Lua. That means that functions can be stored in variables, passed as arguments to other functions, and returned as results. Lua can call (and manipulate) functions written in Lua and functions written in C (see [2.5.7](#)).

The type *userdata* is provided to allow arbitrary C data to be stored in Lua variables. This type corresponds to a block of raw memory and has no pre-defined operations in Lua, except assignment and identity test. However, by using *metatables*, the programmer can define operations for userdata values (see [2.8](#)). Userdata values cannot be created or modified in Lua, only through the C API. This guarantees the integrity of data owned by the host program.

The type *thread* represents independent threads of execution and it is used to implement coroutines.

The type *table* implements associative arrays, that is, arrays that can be indexed not only with numbers, but with any value (except **nil**). Moreover,

tables can be *heterogeneous*, that is, they can contain values of all types (except **nil**). Tables are the sole data structuring mechanism in Lua; they may be used to represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc. To represent records, Lua uses the field name as an index. The language supports this representation by providing `a.name` as syntactic sugar for `a["name"]`. There are several convenient ways to create tables in Lua (see 2.5.6).

Like indices, the value of a table field can be of any type (except **nil**). In particular, because functions are first class values, table fields may contain functions. Thus tables may also carry *methods* (see 2.5.8).

Tables, functions, and userdata values are *objects*: variables do not actually *contain* these values, only *references* to them. Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy.

The library function `type` returns a string describing the type of a given value (see 5.1).

2.2.1 - Coercion

Lua provides automatic conversion between string and number values at run time. Any arithmetic operation applied to a string tries to convert that string to a number, following the usual rules. Conversely, whenever a number is used where a string is expected, the number is converted to a string, in a reasonable format. For complete control of how numbers are converted to strings, use the `format` function from the string library (see 5.3).

2.3 - Variables

Variables are places that store values. There are three kinds of variables in Lua: global variables, local variables, and table fields.

A single name can denote a global variable or a local variable (or a formal parameter of a function, which is a particular form of local variable):

```
var ::= Name
```

Variables are assumed to be global unless explicitly declared local (see 2.4.7). Local variables are *lexically scoped*: Local variables can be freely accessed by functions defined inside their scope (see 2.6).

Before the first assignment to a variable, its value is **nil**.

Square brackets are used to index a table:

```
var ::= prefixexp `[´ exp `]´
```

The first expression (*prefixexp*)should result in a table value; the second expression (*exp*) identifies a specific entry inside that table. The expression denoting the table to be indexed has a restricted syntax; see 2.5 for details.

The syntax `var.NAME` is just syntactic sugar for `var["NAME"]`:

```
var ::= prefixexp `.´ Name
```

The meaning of accesses to global variables and table fields can be changed via metatables. An access to an indexed variable `t[i]` is equivalent to a call `gettable_event(t,i)`. (See 2.8 for a complete description of the `gettable_event` function. This function is not defined or callable in Lua. We use it here only for explanatory purposes.)

All global variables live as fields in ordinary Lua tables, called *environment tables* or simply *environments*. Functions written in C and

exported to Lua (*C functions*) all share a common *global environment*. Each function written in Lua (a *Lua function*) has its own reference to an environment, so that all global variables in that function will refer to that environment table. When a function is created, it inherits the environment from the function that created it. To change or get the environment table of a Lua function, you call `setfenv` or `getfenv` (see [5.1](#)).

An access to a global variable `x` is equivalent to `_env.x`, which in turn is equivalent to

```
gettable_event(_env, "x")
```

where `_env` is the environment of the running function. (The `_env` variable is not defined in Lua. We use it here only for explanatory purposes.)

2.4 - Statements

Lua supports an almost conventional set of statements, similar to those in Pascal or C. This set includes assignment, control structures, procedure calls, table constructors, and variable declarations.

2.4.1 - Chunks

The unit of execution of Lua is called a *chunk*. A chunk is simply a sequence of statements, which are executed sequentially. Each statement can be optionally followed by a semicolon:

```
chunk ::= {stat [`;´]}
```

Lua handles a chunk as the body of an anonymous function (see [2.5.8](#)). As such, chunks can define local variables and return values.

A chunk may be stored in a file or in a string inside the host program. When a chunk is executed, first it is pre-compiled into opcodes for a virtual machine, and then the compiled code is executed by an interpreter for the virtual machine.

Chunks may also be pre-compiled into binary form; see program `luac` for details. Programs in source and compiled forms are interchangeable; Lua automatically detects the file type and acts accordingly.

2.4.2 - Blocks

A block is a list of statements; syntactically, a block is equal to a chunk:

```
block ::= chunk
```

A block may be explicitly delimited to produce a single statement:

```
stat ::= do block end
```

Explicit blocks are useful to control the scope of variable declarations. Explicit blocks are also sometimes used to add a **return** or **break**

statement in the middle of another block (see [2.4.4](#)).

2.4.3 - Assignment

Lua allows multiple assignment. Therefore, the syntax for assignment defines a list of variables on the left side and a list of expressions on the right side. The elements in both lists are separated by commas:

```
stat ::= varlist1 `=´ explist1
varlist1 ::= var {`,´ var}
explist1 ::= exp {`,´ exp}
```

Expressions are discussed in [2.5](#).

Before the assignment, the list of values is *adjusted* to the length of the list of variables. If there are more values than needed, the excess values are thrown away. If there are fewer values than needed, the list is extended with as many **nil**'s as needed. If the list of expressions ends with a function call, then all values returned by that function call enter in the list of values, before the adjustment (except when the call is enclosed in parentheses; see [2.5](#)).

The assignment statement first evaluates all its expressions and only then are the assignments performed. Thus the code

```
i = 3
i, a[i] = i+1, 20
```

sets `a[3]` to 20, without affecting `a[4]` because the `i` in `a[i]` is evaluated (to 3) before it is assigned 4. Similarly, the line

```
x, y = y, x
```

exchanges the values of `x` and `y`.

The meaning of assignments to global variables and table fields can be changed via metatables. An assignment to an indexed variable `t[i] = val` is equivalent to `settable_event(t,i,val)`. (See [2.8](#) for a complete description of the `settable_event` function. This function is not defined or callable in Lua. We use it here only for explanatory purposes.)

An assignment to a global variable `x = val` is equivalent to the assignment `_env.x = val`, which in turn is equivalent to

```
settable_event(_env, "x", val)
```

where `_env` is the environment of the running function. (The `_env` variable is not defined in Lua. We use it here only for explanatory purposes.)

2.4.4 - Control Structures

The control structures **if**, **while**, and **repeat** have the usual meaning and familiar syntax:

```
stat ::= while exp do block end
stat ::= repeat block until exp
stat ::= if exp then block {elseif exp then block} [else blo
```

Lua also has a **for** statement, in two flavors (see 2.4.5).

The condition expression *exp* of a control structure may return any value. Both **false** and **nil** are considered false. All values different from **nil** and **false** are considered true (in particular, the number 0 and the empty string are also true).

The **return** statement is used to return values from a function or from a chunk. Functions and chunks may return more than one value, so the syntax for the **return** statement is

```
stat ::= return [explist1]
```

The **break** statement can be used to terminate the execution of a **while**, **repeat**, or **for** loop, skipping to the next statement after the loop:

```
stat ::= break
```

A **break** ends the innermost enclosing loop.

For syntactic reasons, **return** and **break** statements can only be written as the *last* statement of a block. If it is really necessary to **return** or **break** in the middle of a block, then an explicit inner block can be used, as in the idioms `do return end` and `do break end`, because now **return**

and **break** are the last statements in their (inner) blocks. In practice, those idioms are only used during debugging.

2.4.5 - For Statement

The **for** statement has two forms: one numeric and one generic.

The numeric **for** loop repeats a block of code while a control variable runs through an arithmetic progression. It has the following syntax:

```
stat ::= for Name `=´ exp `,´ exp [`,´ exp] do block end
```

The *block* is repeated for *name* starting at the value of the first *exp*, until it passes the second *exp* by steps of the third *exp*. More precisely, a **for** statement like

```
for var = e1, e2, e3 do block end
```

is equivalent to the code:

```
do
  local var, _limit, _step = tonumber(e1), tonumber(e2), tonu
  if not (var and _limit and _step) then error() end
  while (_step>0 and var<=_limit) or (_step<=0 and var>=_limi
    block
    var = var + _step
  end
end
```

Note the following:

- All three control expressions are evaluated only once, before the loop starts. They must all result in numbers.
- `_limit` and `_step` are invisible variables. The names are here for explanatory purposes only.
- The behavior is *undefined* if you assign to `var` inside the block.
- If the third expression (the step) is absent, then a step of 1 is used.
- You can use **break** to exit a **for** loop.
- The loop variable `var` is local to the statement; you cannot use its value after the **for** ends or is broken. If you need the value of the loop variable `var`, then assign it to another variable before breaking

or exiting the loop.

The generic **for** statement works over functions, called *iterators*. For each iteration, it calls its iterator function to produce a new value, stopping when the new value is **nil**. The generic **for** loop has the following syntax:

```
stat ::= for Name {`,´ Name} in explist1 do block end
```

A **for** statement like

```
for var_1, ..., var_n in explist do block end
```

is equivalent to the code:

```
do
  local _f, _s, var_1 = explist
  local var_2, ... , var_n
  while true do
    var_1, ..., var_n = _f(_s, var_1)
    if var_1 == nil then break end
    block
  end
end
```

Note the following:

- `explist` is evaluated only once. Its results are an *iterator* function, a *state*, and an initial value for the first *iterator variable*.
- `_f` and `_s` are invisible variables. The names are here for explanatory purposes only.
- The behavior is *undefined* if you assign to `var_1` inside the block.
- You can use **break** to exit a **for** loop.
- The loop variables `var_i` are local to the statement; you cannot use their values after the **for** ends. If you need these values, then assign them to other variables before breaking or exiting the loop.

2.4.6 - Function Calls as Statements

To allow possible side-effects, function calls can be executed as statements:

```
stat ::= functioncall
```

In this case, all returned values are thrown away. Function calls are explained in [2.5.7](#).

2.4.7 - Local Declarations

Local variables may be declared anywhere inside a block. The declaration may include an initial assignment:

```
stat ::= local namelist [`=´ explist1]
namelist ::= Name {`,´ Name}
```

If present, an initial assignment has the same semantics of a multiple assignment (see [2.4.3](#)). Otherwise, all variables are initialized with **nil**.

A chunk is also a block (see [2.4.1](#)), so local variables can be declared in a chunk outside any explicit block. Such local variables die when the chunk ends.

The visibility rules for local variables are explained in [2.6](#).

2.5 - Expressions

The basic expressions in Lua are the following:

```
exp ::= prefixexp
exp ::= nil | false | true
exp ::= Number
exp ::= Literal
exp ::= function
exp ::= tableconstructor
prefixexp ::= var | functioncall | `(´ exp `)´
```

Numbers and literal strings are explained in 2.1; variables are explained in 2.3; function definitions are explained in 2.5.8; function calls are explained in 2.5.7; table constructors are explained in 2.5.6.

An expression enclosed in parentheses always results in only one value. Thus, `(f(x,y,z))` is always a single value, even if f returns several values. (The value of `(f(x,y,z))` is the first value returned by f or **nil** if f does not return any values.)

Expressions can also be built with arithmetic operators, relational operators, and logical operators, all of which are explained below.

2.5.1 - Arithmetic Operators

Lua supports the usual arithmetic operators: the binary + (addition), - (subtraction), * (multiplication), / (division), and ^ (exponentiation); and unary - (negation). If the operands are numbers, or strings that can be converted to numbers (see 2.2.1), then all operations except exponentiation have the usual meaning. Exponentiation calls a global function __pow; otherwise, an appropriate metamethod is called (see 2.8). The standard mathematical library defines function __pow, giving the expected meaning to exponentiation (see 5.5).

2.5.2 - Relational Operators

The relational operators in Lua are

```
        ==      ~=     <      >      <=     >=
```

These operators always result in **false** or **true**.

Equality (==) first compares the type of its operands. If the types are different, then the result is **false**. Otherwise, the values of the operands are compared. Numbers and strings are compared in the usual way. Objects (tables, userdata, threads, and functions) are compared by *reference*: Two objects are considered equal only if they are the *same* object. Every time you create a new object (a table, userdata, or function), this new object is different from any previously existing object.

You can change the way that Lua compares tables and userdata using the "eq" metamethod (see 2.8).

The conversion rules of 2.2.1 *do not* apply to equality comparisons. Thus, "0"==0 evaluates to **false**, and t[0] and t["0"] denote different entries in a table.

The operator ~= is exactly the negation of equality (==).

The order operators work as follows. If both arguments are numbers, then they are compared as such. Otherwise, if both arguments are strings, then their values are compared according to the current locale. Otherwise, Lua tries to call the "lt" or the "le" metamethod (see 2.8).

2.5.3 - Logical Operators

The logical operators in Lua are

```
        and     or      not
```

Like the control structures (see 2.4.4), all logical operators consider both **false** and **nil** as false and anything else as true.

The operator **not** always returns **false** or **true**.

The conjunction operator **and** returns its first argument if this value is **false** or **nil**; otherwise, **and** returns its second argument. The disjunction operator **or** returns its first argument if this value is different from **nil** and

false; otherwise, **or** returns its second argument. Both **and** and **or** use short-cut evaluation, that is, the second operand is evaluated only if necessary. For example,

```
10 or error()       -> 10
nil or "a"          -> "a"
nil and 10          -> nil
false and error()   -> false
false and nil       -> false
false or nil        -> nil
10 and 20           -> 20
```

2.5.4 - Concatenation

The string concatenation operator in Lua is denoted by two dots (`` `..´ ``). If both operands are strings or numbers, then they are converted to strings according to the rules mentioned in 2.2.1. Otherwise, the "concat" metamethod is called (see 2.8).

2.5.5 - Precedence

Operator precedence in Lua follows the table below, from lower to higher priority:

```
or
and
<       >       <=      >=      ~=      ==
..
+       -
*       /
not     - (unary)
^
```

You can use parentheses to change the precedences in an expression. The concatenation (`` `..´ ``) and exponentiation (`` `^´ ``) operators are right associative. All other binary operators are left associative.

2.5.6 - Table Constructors

Table constructors are expressions that create tables. Every time a constructor is evaluated, a new table is created. Constructors can be

used to create empty tables, or to create a table and initialize some of its fields. The general syntax for constructors is

```
tableconstructor ::= `{´ [fieldlist] `}´
fieldlist ::= field {fieldsep field} [fieldsep]
field ::= `[´ exp `]´ `=´ exp | Name `=´ exp | exp
fieldsep ::= `,´ | `;´
```

Each field of the form `[exp1]` = `exp2` adds to the new table an entry with key `exp1` and value `exp2`. A field of the form `name` = `exp` is equivalent to `["name"]` = `exp`. Finally, fields of the form `exp` are equivalent to `[i]` = `exp`, where `i` are consecutive numerical integers, starting with 1. Fields in the other formats do not affect this counting. For example,

```
a = {[f(1)] = g; "x", "y"; x = 1, f(x), [30] = 23; 45}
```

is equivalent to

```
do
  local temp = {}
  temp[f(1)] = g
  temp[1] = "x"            -- 1st exp
  temp[2] = "y"            -- 2nd exp
  temp.x = 1               -- temp["x"] = 1
  temp[3] = f(x)           -- 3rd exp
  temp[30] = 23
  temp[4] = 45             -- 4th exp
  a = temp
end
```

If the last field in the list has the form `exp` and the expression is a function call, then all values returned by the call enter the list consecutively (see 2.5.7). To avoid this, enclose the function call in parentheses (see 2.5).

The field list may have an optional trailing separator, as a convenience for machine-generated code.

2.5.7 - Function Calls

A function call in Lua has the following syntax:

```
functioncall ::= prefixexp args
```

In a function call, first *prefixexp* and *args* are evaluated. If the value of *prefixexp* has type *function*, then that function is called with the given arguments. Otherwise, its "call" metamethod is called, having as first parameter the value of *prefixexp*, followed by the original call arguments (see 2.8).

The form

```
functioncall ::= prefixexp `:´ Name args
```

can be used to call "methods". A call `v:name(...)` is syntactic sugar for `v.name(v,...)`, except that `v` is evaluated only once.

Arguments have the following syntax:

```
args ::= `(´ [explist1] `)´
args ::= tableconstructor
args ::= Literal
```

All argument expressions are evaluated before the call. A call of the form `f{...}` is syntactic sugar for `f({...})`, that is, the argument list is a single new table. A call of the form `f'...'` (or `f"..."` or `f[[...]]`) is syntactic sugar for `f('...')`, that is, the argument list is a single literal string.

Because a function can return any number of results (see 2.4.4), the number of results must be adjusted before they are used. If the function is called as a statement (see 2.4.6), then its return list is adjusted to zero elements, thus discarding all returned values. If the function is called inside another expression or in the middle of a list of expressions, then its return list is adjusted to one element, thus discarding all returned values except the first one. If the function is called as the last element of a list of expressions, then no adjustment is made (unless the call is enclosed in parentheses).

Here are some examples:

```
f()                 -- adjusted to 0 results
g(f(), x)           -- f() is adjusted to 1 result
g(x, f())           -- g gets x plus all values returned by f(
a,b,c = f(), x      -- f() is adjusted to 1 result (and c gets
a,b,c = x, f()      -- f() is adjusted to 2 results
```

```
a,b,c = f()          -- f() is adjusted to 3 results
return f()           -- returns all values returned by f()
return x,y,f()       -- returns x, y, and all values returned b
{f()}                -- creates a list with all values returned
{f(), nil}           -- f() is adjusted to 1 result
```

If you enclose a function call in parentheses, then it is adjusted to return exactly one value:

```
return x,y,(f())     -- returns x, y, and the first value from
{(f())}              -- creates a table with exactly one elemen
```

As an exception to the free-format syntax of Lua, you cannot put a line break before the `(´ in a function call. That restriction avoids some ambiguities in the language. If you write

```
a = f
(g).x(a)
```

Lua would read that as `a = f(g).x(a)`. So, if you want two statements, you must add a semi-colon between them. If you actually want to call `f`, you must remove the line break before `(g)`.

A call of the form `return` *functioncall* is called a *tail call*. Lua implements *proper tail calls* (or *proper tail recursion*): In a tail call, the called function reuses the stack entry of the calling function. Therefore, there is no limit on the number of nested tail calls that a program can execute. However, a tail call erases any debug information about the calling function. Note that a tail call only happens with a particular syntax, where the **return** has one single function call as argument; this syntax makes the calling function returns exactly the returns of the called function. So, all the following examples are not tail calls:

```
return (f(x))        -- results adjusted to 1
return 2 * f(x)
return x, f(x)       -- additional results
f(x); return         -- results discarded
return x or f(x)     -- results adjusted to 1
```

2.5.8 - Function Definitions

The syntax for function definition is

```
function ::= function funcbody
funcbody ::= `(´ [parlist1] `)´ block end
```

The following syntactic sugar simplifies function definitions:

```
stat ::= function funcname funcbody
stat ::= local function Name funcbody
funcname ::= Name {`.´ Name} [`:´ Name]
```

The statement

```
function f () ... end
```

translates to

```
f = function () ... end
```

The statement

```
function t.a.b.c.f () ... end
```

translates to

```
t.a.b.c.f = function () ... end
```

The statement

```
local function f () ... end
```

translates to

```
local f; f = function () ... end
```

A function definition is an executable expression, whose value has type *function*. When Lua pre-compiles a chunk, all its function bodies are pre-compiled too. Then, whenever Lua executes the function definition, the function is *instantiated* (or *closed*). This function instance (or *closure*) is the final value of the expression. Different instances of the same function may refer to different external local variables and may have different environment tables.

Parameters act as local variables that are initialized with the argument

values:

```
parlist1 ::= namelist [`,´ `...´]
parlist1 ::= `...´
```

When a function is called, the list of arguments is adjusted to the length of the list of parameters, unless the function is a variadic or *vararg function*, which is indicated by three dots (`...´) at the end of its parameter list. A vararg function does not adjust its argument list; instead, it collects all extra arguments into an implicit parameter, called `arg`. The value of `arg` is a table, with a field `n´ that holds the number of extra arguments and with the extra arguments at positions 1, 2, ..., n.

As an example, consider the following definitions:

```
function f(a, b) end
function g(a, b, ...) end
function r() return 1,2,3 end
```

Then, we have the following mapping from arguments to parameters:

```
CALL                PARAMETERS

f(3)                a=3, b=nil
f(3, 4)             a=3, b=4
f(3, 4, 5)          a=3, b=4
f(r(), 10)          a=1, b=10
f(r())              a=1, b=2

g(3)                a=3, b=nil, arg={n=0}
g(3, 4)             a=3, b=4,   arg={n=0}
g(3, 4, 5, 8)       a=3, b=4,   arg={5, 8; n=2}
g(5, r())           a=5, b=1,   arg={2, 3; n=2}
```

Results are returned using the **return** statement (see 2.4.4). If control reaches the end of a function without encountering a **return** statement, then the function returns with no results.

The *colon* syntax is used for defining *methods*, that is, functions that have an implicit extra parameter `self`. Thus, the statement

```
function t.a.b.c:f (...) ... end
```

is syntactic sugar for

```
t.a.b.c.f = function (self, ...) ... end
```

2.6 - Visibility Rules

Lua is a lexically scoped language. The scope of variables begins at the first statement *after* their declaration and lasts until the end of the innermost block that includes the declaration. For instance:

```
x = 10                    -- global variable
do                        -- new block
  local x = x             -- new `x', with value 10
  print(x)                --> 10
  x = x+1
  do                      -- another block
    local x = x+1         -- another `x'
    print(x)              --> 12
  end
  print(x)                --> 11
end
print(x)                  --> 10  (the global one)
```

Notice that, in a declaration like `local x = x`, the new `x` being declared is not in scope yet, and so the second `x` refers to the outside variable.

Because of the lexical scoping rules, local variables can be freely accessed by functions defined inside their scope. For instance:

```
local counter = 0
function inc (x)
  counter = counter + x
  return counter
end
```

A local variable used by an inner function is called an *upvalue*, or *external local variable*, inside the inner function.

Notice that each execution of a **local** statement defines new local variables. Consider the following example:

```
a = {}
local x = 20
for i=1,10 do
  local y = 0
  a[i] = function () y=y+1; return x+y end
```

```
    end
```

The loop creates ten closures (that is, ten instances of the anonymous function). Each of these closures uses a different y variable, while all of them share the same x.

2.7 - Error Handling

Because Lua is an extension language, all Lua actions start from C code in the host program calling a function from the Lua library (see 3.15). Whenever an error occurs during Lua compilation or execution, control returns to C, which can take appropriate measures (such as print an error message).

Lua code can explicitly generate an error by calling the `error` function (see 5.1). If you need to catch errors in Lua, you can use the `pcall` function (see 5.1).

2.8 - Metatables

Every table and userdata object in Lua may have a *metatable*. This *metatable* is an ordinary Lua table that defines the behavior of the original table and userdata under certain special operations. You can change several aspects of the behavior of an object by setting specific fields in its metatable. For instance, when an object is the operand of an addition, Lua checks for a function in the field `"__add"` in its metatable. If it finds one, Lua calls that function to perform the addition.

We call the keys in a metatable *events* and the values *metamethods*. In the previous example, the event is `"add"` and the metamethod is the function that performs the addition.

You can query and change the metatable of an object through the `set/getmetatable` functions (see [5.1](#)).

A metatable may control how an object behaves in arithmetic operations, order comparisons, concatenation, and indexing. A metatable can also define a function to be called when a userdata is garbage collected. For each of those operations Lua associates a specific key called an *event*. When Lua performs one of those operations over a table or a userdata, it checks whether that object has a metatable with the corresponding event. If so, the value associated with that key (the *metamethod*) controls how Lua will perform the operation.

Metatables control the operations listed next. Each operation is identified by its corresponding name. The key for each operation is a string with its name prefixed by two underscores; for instance, the key for operation "add" is the string `"__add"`. The semantics of these operations is better explained by a Lua function describing how the interpreter executes that operation.

The code shown here in Lua is only illustrative; the real behavior is hard coded in the interpreter and it is much more efficient than this simulation. All functions used in these descriptions (`rawget`, `tonumber`, etc.) are

described in . In particular, to retrieve the metamethod of a given object, we use the expression

```
metatable(obj)[event]
```

This should be read as

```
rawget(metatable(obj) or {}, event)
```

That is, the access to a metamethod does not invoke other metamethods, and the access to objects with no metatables does not fail (it simply results in **nil**).

- **"add":** the + operation.

  The function `getbinhandler` below defines how Lua chooses a handler for a binary operation. First, Lua tries the first operand. If its type does not define a handler for the operation, then Lua tries the second operand.

  ```
  function getbinhandler (op1, op2, event)
    return metatable(op1)[event] or metatable(op2)[event]
  end
  ```

  Using that function, the behavior of the op1 + op2 is

  ```
  function add_event (op1, op2)
    local o1, o2 = tonumber(op1), tonumber(op2)
    if o1 and o2 then  -- both operands are numeric?
      return o1 + o2   -- `+' here is the primitive `add'
    else  -- at least one of the operands is not numeric
      local h = getbinhandler(op1, op2, "__add")
      if h then
        -- call the handler with both operands
        return h(op1, op2)
      else  -- no handler available: default behavior
        error("...")
      end
    end
  end
  ```

- **"sub":** the - operation. Behavior similar to the "add" operation.

- **"mul":** the * operation. Behavior similar to the "add" operation.

- **"div":** the / operation. Behavior similar to the "add" operation.

- **"pow":** the ^ (exponentiation) operation.

```
function pow_event (op1, op2)
  local o1, o2 = tonumber(op1), tonumber(op2)
  if o1 and o2 then  -- both operands are numeric?
    return __pow(o1, o2)   -- call global `__pow'
  else  -- at least one of the operands is not numeric
    local h = getbinhandler(op1, op2, "__pow")
    if h then
      -- call the handler with both operands
      return h(op1, op2)
    else  -- no handler available: default behavior
      error("...")
    end
  end
end
```

- **"unm":** the unary - operation.

```
function unm_event (op)
  local o = tonumber(op)
  if o then  -- operand is numeric?
    return -o  -- `-' here is the primitive `unm'
  else  -- the operand is not numeric.
    -- Try to get a handler from the operand
    local h = metatable(op).__unm
    if h then
      -- call the handler with the operand and nil
      return h(op, nil)
    else  -- no handler available: default behavior
      error("...")
    end
  end
end
```

- **"concat":** the .. (concatenation) operation.

```
function concat_event (op1, op2)
  if (type(op1) == "string" or type(op1) == "number") and
     (type(op2) == "string" or type(op2) == "number") then
    return op1 .. op2  -- primitive string concatenation
  else
    local h = getbinhandler(op1, op2, "__concat")
    if h then
```

```
      return h(op1, op2)
    else
      error("...")
    end
  end
end
```

- **"eq":** the `==` operation. The function `getcomphandler` defines how Lua chooses a metamethod for comparison operators. A metamethod only is selected when both objects being compared have the same type and the same metamethod for the selected operation.

  ```
  function getcomphandler (op1, op2, event)
    if type(op1) ~= type(op2) then return nil end
    local mm1 = metatable(op1)[event]
    local mm2 = metatable(op2)[event]
    if mm1 == mm2 then return mm1 else return nil end
  end
  ```

  The "eq" event is defined as follows:

  ```
  function eq_event (op1, op2)
    if type(op1) ~= type(op2) then   -- different types?
      return false    -- different objects
    end
    if op1 == op2 then    -- primitive equal?
      return true    -- objects are equal
    end
    -- try metamethod
    local h = getcomphandler(op1, op2, "__eq")
    if h then
      return h(op1, op2)
    else
      return false
    end
  end
  ```

  `a ~= b` is equivalent to `not (a == b)`.

- **"lt":** the `<` operation.

  ```
  function lt_event (op1, op2)
    if type(op1) == "number" and type(op2) == "number" then
      return op1 < op2   -- numeric comparison
    elseif type(op1) == "string" and type(op2) == "string" then
      return op1 < op2   -- lexicographic comparison
  ```

```
        else
          local h = getcomphandler(op1, op2, "__lt")
          if h then
            return h(op1, op2)
          else
            error("...");
          end
        end
      end
```

`a > b` is equivalent to `b < a`.

- **"le":** the `<=` operation.

```
  function le_event (op1, op2)
    if type(op1) == "number" and type(op2) == "number" then
      return op1 <= op2    -- numeric comparison
    elseif type(op1) == "string" and type(op2) == "string" then
      return op1 <= op2    -- lexicographic comparison
    else
      local h = getcomphandler(op1, op2, "__le")
      if h then
        return h(op1, op2)
      else
        h = getcomphandler(op1, op2, "__lt")
        if h then
          return not h(op2, op1)
        else
          error("...");
        end
      end
    end
  end
```

`a >= b` is equivalent to `b <= a`. Note that, in the absence of a "le"
metamethod, Lua tries the "lt", assuming that `a <= b` is equivalent to
`not (b < a)`.

- **"index":** The indexing access `table[key]`.

```
  function gettable_event (table, key)
    local h
    if type(table) == "table" then
      local v = rawget(table, key)
      if v ~= nil then return v end
      h = metatable(table).__index
```

```
      if h == nil then return nil end
    else
      h = metatable(table).__index
      if h == nil then
        error("...");
      end
    end
    if type(h) == "function" then
      return h(table, key)      -- call the handler
    else return h[key]          -- or repeat operation on it
  end
```

- **"newindex":** The indexing assignment `table[key] = value`.

```
function settable_event (table, key, value)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    if v ~= nil then rawset(table, key, value); return end
    h = metatable(table).__newindex
    if h == nil then rawset(table, key, value); return end
  else
    h = metatable(table).__newindex
    if h == nil then
      error("...");
    end
  end
  if type(h) == "function" then
    return h(table, key,value)    -- call the handler
  else h[key] = value             -- or repeat operation on it
end
```

- **"call":** called when Lua calls a value.

```
function function_event (func, ...)
  if type(func) == "function" then
    return func(unpack(arg))   -- primitive call
  else
    local h = metatable(func).__call
    if h then
      return h(func, unpack(arg))
    else
      error("...")
    end
  end
end
```

2.9 - Garbage Collection

Lua does automatic memory management. That means that you do not have to worry about allocating memory for new objects and freeing it when the objects are no longer needed. Lua manages memory automatically by running a *garbage collector* from time to time to collect all *dead objects* (that is, those objects that are no longer accessible from Lua). All objects in Lua are subject to automatic management: tables, userdata, functions, threads, and strings.

Lua uses two numbers to control its garbage-collection cycles. One number counts how many bytes of dynamic memory Lua is using; the other is a threshold. When the number of bytes crosses the threshold, Lua runs the garbage collector, which reclaims the memory of all dead objects. The byte counter is adjusted, and then the threshold is reset to twice the new value of the byte counter.

Through the C API, you can query those numbers and change the threshold (see [3.7](#)). Setting the threshold to zero actually forces an immediate garbage-collection cycle, while setting it to a huge number effectively stops the garbage collector. Using Lua code you have a more limited control over garbage-collection cycles, through the `gcinfo` and `collectgarbage` functions (see [5.1](#)).

2.9.1 - Garbage-Collection Metamethods

Using the C API, you can set garbage-collector metamethods for userdata (see [2.8](#)). These metamethods are also called *finalizers*. Finalizers allow you to coordinate Lua's garbage collection with external resource management (such as closing files, network or database connections, or freeing your own memory).

Free userdata with a field `__gc` in their metatables are not collected immediately by the garbage collector. Instead, Lua puts them in a list. After the collection, Lua does the equivalent of the following function for each userdata in that list:

```
function gc_event (udata)
  local h = metatable(udata).__gc
  if h then
    h(udata)
  end
end
```

At the end of each garbage-collection cycle, the finalizers for userdata are called in *reverse* order of their creation, among those collected in that cycle. That is, the first finalizer to be called is the one associated with the userdata created last in the program.

2.9.2 - Weak Tables

A *weak table* is a table whose elements are *weak references*. A weak reference is ignored by the garbage collector. In other words, if the only references to an object are weak references, then the garbage collector will collect that object.

A weak table can have weak keys, weak values, or both. A table with weak keys allows the collection of its keys, but prevents the collection of its values. A table with both weak keys and weak values allows the collection of both keys and values. In any case, if either the key or the value is collected, the whole pair is removed from the table. The weakness of a table is controlled by the value of the `__mode` field of its metatable. If the `__mode` field is a string containing the character `k´, the keys in the table are weak. If `__mode` contains `v´, the values in the table are weak.

After you use a table as a metatable, you should not change the value of its field `__mode`. Otherwise, the weak behavior of the tables controlled by this metatable is undefined.

2.10 - Coroutines

Lua supports coroutines, also called *semi-coroutines* or *collaborative multithreading*. A coroutine in Lua represents an independent thread of execution. Unlike threads in multithread systems, however, a coroutine only suspends its execution by explicitly calling a yield function.

You create a coroutine with a call to `coroutine.create`. Its sole argument is a function that is the main function of the coroutine. The `create` function only creates a new coroutine and returns a handle to it (an object of type *thread*); it does not start the coroutine execution.

When you first call `coroutine.resume`, passing as its first argument the thread returned by `coroutine.create`, the coroutine starts its execution, at the first line of its main function. Extra arguments passed to `coroutine.resume` are given as parameters for the coroutine main function. After the coroutine starts running, it runs until it terminates or *yields*.

A coroutine can terminate its execution in two ways: Normally, when its main function returns (explicitly or implicitly, after the last instruction); and abnormally, if there is an unprotected error. In the first case, `coroutine.resume` returns **true**, plus any values returned by the coroutine main function. In case of errors, `coroutine.resume` returns **false** plus an error message.

A coroutine yields by calling `coroutine.yield`. When a coroutine yields, the corresponding `coroutine.resume` returns immediately, even if the yield happens inside nested function calls (that is, not in the main function, but in a function directly or indirectly called by the main function). In the case of a yield, `coroutine.resume` also returns **true**, plus any values passed to `coroutine.yield`. The next time you resume the same coroutine, it continues its execution from the point where it yielded, with the call to `coroutine.yield` returning any extra arguments passed to `coroutine.resume`.

The `coroutine.wrap` function creates a coroutine like `coroutine.create`, but instead of returning the coroutine itself, it returns a function that, when called, resumes the coroutine. Any arguments passed to that function go as extra arguments to resume. The function returns all the values returned by resume, except the first one (the boolean error code). Unlike `coroutine.resume`, this function does not catch errors; any error is propagated to the caller.

As an example, consider the next code:

```
function foo1 (a)
  print("foo", a)
  return coroutine.yield(2*a)
end

co = coroutine.create(function (a,b)
     print("co-body", a, b)
     local r = foo1(a+1)
     print("co-body", r)
     local r, s = coroutine.yield(a+b, a-b)
     print("co-body", r, s)
     return b, "end"
end)

a, b = coroutine.resume(co, 1, 10)
print("main", a, b)
a, b, c = coroutine.resume(co, "r")
print("main", a, b, c)
a, b, c = coroutine.resume(co, "x", "y")
print("main", a, b, c)
a, b = coroutine.resume(co, "x", "y")
print("main", a, b)
```

When you run it, it produces the following output:

```
co-body 1       10
foo     2
main    true    4
co-body r
main    true    11      -9
co-body x       y
main    true    10      end
main    false   cannot resume dead coroutine
```

3 - The Application Program Interface

This section describes the C API for Lua, that is, the set of C functions available to the host program to communicate with Lua. All API functions and related types and constants are declared in the header file `lua.h`.

Even when we use the term "function", any facility in the API may be provided as a *macro* instead. All such macros use each of its arguments exactly once (except for the first argument, which is always a Lua state), and so do not generate hidden side-effects.

3.1 - States

The Lua library is fully reentrant: it has no global variables. The whole state of the Lua interpreter (global variables, stack, etc.) is stored in a dynamically allocated structure of type `lua_State`. A pointer to this state must be passed as the first argument to every function in the library, except to `lua_open`, which creates a Lua state from scratch.

Before calling any API function, you must create a state by calling `lua_open`:

```
lua_State *lua_open (void);
```

To release a state created with `lua_open`, call `lua_close`:

```
void lua_close (lua_State *L);
```

This function destroys all objects in the given Lua state (calling the corresponding garbage-collection metamethods, if any) and frees all dynamic memory used by that state. On several platforms, you may not need to call this function, because all resources are naturally released when the host program ends. On the other hand, long-running programs, such as a daemon or a web server, might need to release states as soon as they are not needed, to avoid growing too large.

3.2 - The Stack and Indices

Lua uses a *virtual stack* to pass values to and from C. Each element in this stack represents a Lua value (**nil**, number, string, etc.).

Whenever Lua calls C, the called function gets a new stack, which is independent of previous stacks and of stacks of C functions that are still active. That stack initially contains any arguments to the C function, and it is where the C function pushes its results to be returned to the caller (see 3.16).

For convenience, most query operations in the API do not follow a strict stack discipline. Instead, they can refer to any element in the stack by using an *index*: A positive index represents an *absolute* stack position (starting at 1); a negative index represents an *offset* from the top of the stack. More specifically, if the stack has *n* elements, then index 1 represents the first element (that is, the element that was pushed onto the stack first) and index *n* represents the last element; index *-1* also represents the last element (that is, the element at the top) and index *-n* represents the first element. We say that an index is *valid* if it lies between 1 and the stack top (that is, if `1 <= abs(index) <= top`).

At any time, you can get the index of the top element by calling `lua_gettop`:

```
int lua_gettop (lua_State *L);
```

Because indices start at 1, the result of `lua_gettop` is equal to the number of elements in the stack (and so 0 means an empty stack).

When you interact with Lua API, *you are responsible for controlling stack overflow*. The function

```
int lua_checkstack (lua_State *L, int extra);
```

grows the stack size to `top + extra` elements; it returns false if it cannot grow the stack to that size. This function never shrinks the stack; if the stack is already larger than the new size, it is left unchanged.

Whenever Lua calls C, it ensures that at least `LUA_MINSTACK` stack positions are available. `LUA_MINSTACK` is defined in `lua.h` as 20, so that usually you do not have to worry about stack space unless your code has loops pushing elements onto the stack.

Most query functions accept as indices any value inside the available stack space, that is, indices up to the maximum stack size you have set through `lua_checkstack`. Such indices are called *acceptable indices*. More formally, we define an *acceptable index* as follows:

```
     (index < 0 && abs(index) <= top) || (index > 0 && index <= stac
```

Note that 0 is never an acceptable index.

Unless otherwise noted, any function that accepts valid indices can also be called with *pseudo-indices*, which represent some Lua values that are accessible to the C code but are not in the stack. Pseudo-indices are used to access the global environment, the registry, and the upvalues of a C function (see 3.17).

3.3 - Stack Manipulation

The API offers the following functions for basic stack manipulation:

```
void lua_settop    (lua_State *L, int index);
void lua_pushvalue (lua_State *L, int index);
void lua_remove    (lua_State *L, int index);
void lua_insert    (lua_State *L, int index);
void lua_replace   (lua_State *L, int index);
```

`lua_settop` accepts any acceptable index, or 0, and sets the stack top to that index. If the new top is larger than the old one, then the new elements are filled with **nil**. If `index` is 0, then all stack elements are removed. A useful macro defined in the `lua.h` is

```
#define lua_pop(L,n)   lua_settop(L, -(n)-1)
```

which pops `n` elements from the stack.

`lua_pushvalue` pushes onto the stack a copy of the element at the given index. `lua_remove` removes the element at the given position, shifting down the elements above that position to fill the gap. `lua_insert` moves the top element into the given position, shifting up the elements above that position to open space. `lua_replace` moves the top element into the given position, without shifting any element (therefore replacing the value at the given position). All these functions accept only valid indices. (You cannot call `lua_remove` or `lua_insert` with pseudo-indices, as they do not represent a stack position.)

As an example, if the stack starts as `10 20 30 40 50*` (from bottom to top; the `*´ marks the top), then

```
lua_pushvalue(L, 3)    --> 10 20 30 40 50 30*
lua_pushvalue(L, -1)   --> 10 20 30 40 50 30 30*
lua_remove(L, -3)      --> 10 20 30 40 30 30*
lua_remove(L,  6)      --> 10 20 30 40 30*
lua_insert(L,  1)      --> 30 10 20 30 40*
lua_insert(L, -1)      --> 30 10 20 30 40*  (no effect)
lua_replace(L, 2)      --> 30 40 20 30*
lua_settop(L, -3)      --> 30 40*
```

```
lua_settop(L,  6)      --> 30 40 nil nil nil nil*
```

3.4 - Querying the Stack

To check the type of a stack element, the following functions are
available:

```
int lua_type              (lua_State *L, int index);
int lua_isnil             (lua_State *L, int index);
int lua_isboolean         (lua_State *L, int index);
int lua_isnumber          (lua_State *L, int index);
int lua_isstring          (lua_State *L, int index);
int lua_istable           (lua_State *L, int index);
int lua_isfunction        (lua_State *L, int index);
int lua_iscfunction       (lua_State *L, int index);
int lua_isuserdata        (lua_State *L, int index);
int lua_islightuserdata   (lua_State *L, int index);
```

These functions can be called with any acceptable index.

`lua_type` returns the type of a value in the stack, or `LUA_TNONE` for a non-
valid index (that is, if that stack position is "empty"). The types returned
by `lua_type` are coded by the following constants defined in `lua.h`:
`LUA_TNIL`, `LUA_TNUMBER`, `LUA_TBOOLEAN`, `LUA_TSTRING`, `LUA_TTABLE`,
`LUA_TFUNCTION`, `LUA_TUSERDATA`, `LUA_TTHREAD`, `LUA_TLIGHTUSERDATA`. The
following function translates these constants to strings:

```
const char *lua_typename  (lua_State *L, int type);
```

The `lua_is*` functions return 1 if the object is compatible with the given
type, and 0 otherwise. `lua_isboolean` is an exception to this rule: It
succeeds only for boolean values (otherwise it would be useless, as any
value has a boolean value). They always return 0 for a non-valid index.
`lua_isnumber` accepts numbers and numerical strings; `lua_isstring`
accepts strings and numbers (see [2.2.1](#)); `lua_isfunction` accepts both
Lua functions and C functions; and `lua_isuserdata` accepts both full and
light userdata. To distinguish between Lua functions and C functions, you
can use `lua_iscfunction`. To distinguish between full and light userdata,
you can use `lua_islightuserdata`. To distinguish between numbers and
numerical strings, you can use `lua_type`.

The API also contains functions to compare two values in the stack:

```
int lua_equal    (lua_State *L, int index1, int index2);
int lua_rawequal (lua_State *L, int index1, int index2);
int lua_lessthan (lua_State *L, int index1, int index2);
```

`lua_equal` and `lua_lessthan` are equivalent to their counterparts in Lua (see [2.5.2](#)). `lua_rawequal` compares the values for primitive equality, without metamethods. These functions return 0 (false) if any of the indices are non-valid.

3.5 - Getting Values from the Stack

To translate a value in the stack to a specific C type, you can use the following conversion functions:

```
int            lua_toboolean   (lua_State *L, int index);
lua_Number     lua_tonumber    (lua_State *L, int index);
const char    *lua_tostring    (lua_State *L, int index);
size_t         lua_strlen      (lua_State *L, int index);
lua_CFunction  lua_tocfunction (lua_State *L, int index);
void          *lua_touserdata  (lua_State *L, int index);
lua_State     *lua_tothread    (lua_State *L, int index);
void          *lua_topointer   (lua_State *L, int index);
```

These functions can be called with any acceptable index. When called with a non-valid index, they act as if the given value had an incorrect type.

`lua_toboolean` converts the Lua value at the given index to a C "boolean" value (0 or 1). Like all tests in Lua, `lua_toboolean` returns 1 for any Lua value different from **false** and **nil**; otherwise it returns 0. It also returns 0 when called with a non-valid index. (If you want to accept only real boolean values, use `lua_isboolean` to test the type of the value.)

`lua_tonumber` converts the Lua value at the given index to a number (by default, `lua_Number` is `double`). The Lua value must be a number or a string convertible to number (see [2.2.1](#)); otherwise, `lua_tonumber` returns 0.

`lua_tostring` converts the Lua value at the given index to a string (`const char*`). The Lua value must be a string or a number; otherwise, the function returns `NULL`. If the value is a number, then `lua_tostring` also *changes the actual value in the stack to a string*. (This change confuses `lua_next` when `lua_tostring` is applied to keys.) `lua_tostring` returns a fully aligned pointer to a string inside the Lua state. This string always has a zero (`'\0'`) after its last character (as in C), but may contain other zeros in its body. If you do not know whether a string may contain zeros, you can use `lua_strlen` to get its actual length. Because Lua has

garbage collection, there is no guarantee that the pointer returned by `lua_tostring` will be valid after the corresponding value is removed from the stack. If you need the string after the current function returns, then you should duplicate it or put it into the registry (see 3.18).

`lua_tocfunction` converts a value in the stack to a C function. This value must be a C function; otherwise, `lua_tocfunction` returns NULL. The type `lua_CFunction` is explained in 3.16.

`lua_tothread` converts a value in the stack to a Lua thread (represented as `lua_State *`). This value must be a thread; otherwise, `lua_tothread` returns NULL.

`lua_topointer` converts a value in the stack to a generic C pointer (`void *`). The value may be a userdata, a table, a thread, or a function; otherwise, `lua_topointer` returns NULL. Lua ensures that different objects of the same type return different pointers. There is no direct way to convert the pointer back to its original value. Typically this function is used for debug information.

`lua_touserdata` is explained in 3.8.

3.6 - Pushing Values onto the Stack

The API has the following functions to push C values onto the stack:

```
void lua_pushboolean       (lua_State *L, int b);
void lua_pushnumber        (lua_State *L, lua_Number n);
void lua_pushlstring       (lua_State *L, const char *s, size
void lua_pushstring        (lua_State *L, const char *s);
void lua_pushnil           (lua_State *L);
void lua_pushcfunction     (lua_State *L, lua_CFunction f);
void lua_pushlightuserdata (lua_State *L, void *p);
```

These functions receive a C value, convert it to a corresponding Lua value, and push the result onto the stack. In particular, `lua_pushlstring` and `lua_pushstring` make an internal copy of the given string. `lua_pushstring` can only be used to push proper C strings (that is, strings that end with a zero and do not contain embedded zeros); otherwise, you should use the more general `lua_pushlstring`, which accepts an explicit size.

You can also push "formatted" strings:

```
const char *lua_pushfstring  (lua_State *L, const char *fmt,
const char *lua_pushvfstring (lua_State *L, const char *fmt,
```

These functions push onto the stack a formatted string and return a pointer to that string. They are similar to `sprintf` and `vsprintf`, but with some important differences:

- You do not have to allocate the space for the result: The result is a Lua string and Lua takes care of memory allocation (and deallocation, through garbage collection).
- The conversion specifiers are quite restricted. There are no flags, widths, or precisions. The conversion specifiers can be simply `` `%%´ `` (inserts a `` `%´ `` in the string), `` `%s´ `` (inserts a zero-terminated string, with no size restrictions), `` `%f´ `` (inserts a `lua_Number`), `` `%d´ `` (inserts an `int`), and `` `%c´ `` (inserts an `int` as a character).

The function

```
void lua_concat (lua_State *L, int n);
```

concatenates the n values at the top of the stack, pops them, and leaves the result at the top. If n is 1, the result is that single string (that is, the function does nothing); if n is 0, the result is the empty string. Concatenation is done following the usual semantics of Lua (see 2.5.4).

3.7 - Controlling Garbage Collection

Lua uses two numbers to control its garbage collection: the *count* and the *threshold* (see 2.9). The first counts the amount of memory in use by Lua; when the count reaches the threshold, Lua runs its garbage collector. After the collection, the count is updated and the threshold is set to twice the count value.

You can access the current values of these two numbers through the following functions:

```
int  lua_getgccount     (lua_State *L);
int  lua_getgcthreshold (lua_State *L);
```

Both return their respective values in Kbytes. You can change the threshold value with

```
void  lua_setgcthreshold (lua_State *L, int newthreshold);
```

Again, the `newthreshold` value is given in Kbytes. When you call this function, Lua sets the new threshold and checks it against the byte counter. If the new threshold is less than the byte counter, then Lua immediately runs the garbage collector. In particular `lua_setgcthreshold(L,0)` forces a garbage collection. After the collection, a new threshold is set according to the previous rule.

3.8 - Userdata

Userdata represents C values in Lua. Lua supports two types of userdata: *full userdata* and *light userdata*.

A full userdata represents a block of memory. It is an object (like a table): You must create it, it can have its own metatable, and you can detect when it is being collected. A full userdata is only equal to itself (under raw equality).

A light userdata represents a pointer. It is a value (like a number): You do not create it, it has no metatables, it is not collected (as it was never created). A light userdata is equal to "any" light userdata with the same C address.

In Lua code, there is no way to test whether a userdata is full or light; both have type `userdata`. In C code, `lua_type` returns `LUA_TUSERDATA` for full userdata, and `LUA_TLIGHTUSERDATA` for light userdata.

You can create a new full userdata with the following function:

```
void *lua_newuserdata (lua_State *L, size_t size);
```

This function allocates a new block of memory with the given size, pushes on the stack a new userdata with the block address, and returns this address.

To push a light userdata into the stack you use `lua_pushlightuserdata` (see [3.6](#)).

`lua_touserdata` (see [3.5](#)) retrieves the value of a userdata. When applied on a full userdata, it returns the address of its block; when applied on a light userdata, it returns its pointer; when applied on a non-userdata value, it returns `NULL`.

When Lua collects a full userdata, it calls the userdata's `gc` metamethod, if any, and then it frees the userdata's corresponding memory.

3.9 - Metatables

The following functions allow you to manipulate the metatables of an object:

```
int lua_getmetatable (lua_State *L, int index);
int lua_setmetatable (lua_State *L, int index);
```

`lua_getmetatable` pushes on the stack the metatable of a given object. If the index is not valid, or if the object does not have a metatable, `lua_getmetatable` returns 0 and pushes nothing on the stack.

`lua_setmetatable` pops a table from the stack and sets it as the new metatable for the given object. `lua_setmetatable` returns 0 when it cannot set the metatable of the given object (that is, when the object is neither a userdata nor a table); even then it pops the table from the stack.

3.10 - Loading Lua Chunks

You can load a Lua chunk with `lua_load`:

```
typedef const char * (*lua_Chunkreader)
                      (lua_State *L, void *data, size_t *s

int lua_load (lua_State *L, lua_Chunkreader reader, void *dat
                      const char *chunkname);
```

The return values of `lua_load` are:

- 0 --- no errors;
- `LUA_ERRSYNTAX` --- syntax error during pre-compilation.
- `LUA_ERRMEM` --- memory allocation error.

If there are no errors, `lua_load` pushes the compiled chunk as a Lua
function on top of the stack. Otherwise, it pushes an error message.

`lua_load` automatically detects whether the chunk is text or binary, and
loads it accordingly (see program `luac`).

`lua_load` uses a user-supplied *reader* function to read the chunk.
Everytime it needs another piece of the chunk, `lua_load` calls the reader,
passing along its `data` parameter. The reader must return a pointer to a
block of memory with a new piece of the chunk and set `size` to the block
size. To signal the end of the chunk, the reader returns `NULL`. The reader
function may return pieces of any size greater than zero.

In the current implementation, the reader function cannot call any Lua
function; to ensure that, it always receives `NULL` as the Lua state.

The *chunkname* is used for error messages and debug information (see
[4](#)).

See the auxiliary library (`lauxlib.c`) for examples of how to use `lua_load`
and for some ready-to-use functions to load chunks from files and strings.

3.11 - Manipulating Tables

Tables are created by calling the function

```
void lua_newtable (lua_State *L);
```

This function creates a new, empty table and pushes it onto the stack.

To read a value from a table that resides somewhere in the stack, call

```
void lua_gettable (lua_State *L, int index);
```

where `index` points to the table. `lua_gettable` pops a key from the stack and returns (on the stack) the contents of the table at that key. The table is left where it was in the stack. As in Lua, this function may trigger a metamethod for the "index" event (see 2.8). To get the real value of any table key, without invoking any metamethod, use the *raw* version:

```
void lua_rawget (lua_State *L, int index);
```

To store a value into a table that resides somewhere in the stack, you push the key and then the value onto the stack, and call

```
void lua_settable (lua_State *L, int index);
```

where `index` points to the table. `lua_settable` pops from the stack both the key and the value. The table is left where it was in the stack. As in Lua, this operation may trigger a metamethod for the "settable" or "newindex" events. To set the real value of any table index, without invoking any metamethod, use the *raw* version:

```
void lua_rawset (lua_State *L, int index);
```

You can traverse a table with the function

```
int lua_next (lua_State *L, int index);
```

where `index` points to the table to be traversed. The function pops a key from the stack, and pushes a key-value pair from the table (the "next" pair after the given key). If there are no more elements, then `lua_next`

returns 0 (and pushes nothing). Use a **nil** key to signal the start of a traversal.

A typical traversal looks like this:

```
/* table is in the stack at index `t' */
lua_pushnil(L);  /* first key */
while (lua_next(L, t) != 0) {
  /* `key' is at index -2 and `value' at index -1 */
  printf("%s - %s\n",
    lua_typename(L, lua_type(L, -2)), lua_typename(L, lua_typ
  lua_pop(L, 1);  /* removes `value'; keeps `key' for next it
}
```

While traversing a table, do not call `lua_tostring` directly on a key, unless you know that the key is actually a string. Recall that `lua_tostring` *changes* the value at the given index; this confuses the next call to `lua_next`.

3.12 - Manipulating Environments

All global variables are kept in ordinary Lua tables, called environments. The initial environment is called the global environment. This table is always at pseudo-index `LUA_GLOBALSINDEX`.

To access and change the value of global variables, you can use regular table operations over an environment table. For instance, to access the value of a global variable, do

```
lua_pushstring(L, varname);
lua_gettable(L, LUA_GLOBALSINDEX);
```

You can change the global environment of a Lua thread using `lua_replace`.

The following functions get and set the environment of Lua functions:

```
void lua_getfenv (lua_State *L, int index);
int  lua_setfenv (lua_State *L, int index);
```

`lua_getfenv` pushes on the stack the environment table of the function at index `index` in the stack. If the function is a C function, `lua_getfenv` pushes the global environment. `lua_setfenv` pops a table from the stack and sets it as the new environment for the function at index `index` in the stack. If the object at the given index is not a Lua function, `lua_setfenv` returns 0.

3.13 - Using Tables as Arrays

The API has functions that help to use Lua tables as arrays, that is,
tables indexed by numbers only:

```
void lua_rawgeti (lua_State *L, int index, int n);
void lua_rawseti (lua_State *L, int index, int n);
```

`lua_rawgeti` pushes the value of the *n*-th element of the table at stack
position `index`. `lua_rawseti` sets the value of the *n*-th element of the table
at stack position `index` to the value at the top of the stack, removing this
value from the stack.

3.14 - Calling Functions

Functions defined in Lua and C functions registered in Lua can be called from the host program. This is done using the following protocol: First, the function to be called is pushed onto the stack; then, the arguments to the function are pushed in *direct order*, that is, the first argument is pushed first. Finally, the function is called using

```
void lua_call (lua_State *L, int nargs, int nresults);
```

`nargs` is the number of arguments that you pushed onto the stack. All arguments and the function value are popped from the stack, and the function results are pushed. The number of results are adjusted to `nresults`, unless `nresults` is `LUA_MULTRET`. In that case, *all* results from the function are pushed. Lua takes care that the returned values fit into the stack space. The function results are pushed onto the stack in direct order (the first result is pushed first), so that after the call the last result is on the top.

The following example shows how the host program may do the equivalent to this Lua code:

```
a = f("how", t.x, 14)
```

Here it is in C:

```
lua_pushstring(L, "t");
lua_gettable(L, LUA_GLOBALSINDEX);          /* global `t' (for l
lua_pushstring(L, "a");                             /*
lua_pushstring(L, "f");                              /* func
lua_gettable(L, LUA_GLOBALSINDEX);             /* function to
lua_pushstring(L, "how");                            /* 1st
lua_pushstring(L, "x");                       /* push the s
lua_gettable(L, -5);                      /* push result of t.x
lua_pushnumber(L, 14);                             /* 3rd
lua_call(L, 3, 1);          /* call function with 3 arguments and
lua_settable(L, LUA_GLOBALSINDEX);            /* set global var
lua_pop(L, 1);                             /* remove `t' from
```

Note that the code above is "balanced": at its end, the stack is back to its

original configuration. This is considered good programming practice.

(We did this example using only the raw functions provided by Lua's API, to show all the details. Usually programmers define and use several macros and auxiliary functions that provide higher level access to Lua. See the source code of the standard libraries for examples.)

3.15 - Protected Calls

When you call a function with `lua_call`, any error inside the called function is propagated upwards (with a `longjmp`). If you need to handle errors, then you should use `lua_pcall`:

```
int lua_pcall (lua_State *L, int nargs, int nresults, int err
```

Both `nargs` and `nresults` have the same meaning as in `lua_call`. If there are no errors during the call, `lua_pcall` behaves exactly like `lua_call`. However, if there is any error, `lua_pcall` catches it, pushes a single value at the stack (the error message), and returns an error code. Like `lua_call`, `lua_pcall` always removes the function and its arguments from the stack.

If `errfunc` is 0, then the error message returned is exactly the original error message. Otherwise, `errfunc` gives the stack index for an *error handler function*. (In the current implementation, that index cannot be a pseudo-index.) In case of runtime errors, that function will be called with the error message and its return value will be the message returned by `lua_pcall`.

Typically, the error handler function is used to add more debug information to the error message, such as a stack traceback. Such information cannot be gathered after the return of `lua_pcall`, since by then the stack has unwound.

The `lua_pcall` function returns 0 in case of success or one of the following error codes (defined in `lua.h`):

- `LUA_ERRRUN` --- a runtime error.
- `LUA_ERRMEM` --- memory allocation error. For such errors, Lua does not call the error handler function.
- `LUA_ERRERR` --- error while running the error handler function.

3.16 - Defining C Functions

Lua can be extended with functions written in C. These functions must be of type `lua_CFunction`, which is defined as

```
typedef int (*lua_CFunction) (lua_State *L);
```

A C function receives a Lua state and returns an integer, the number of values it wants to return to Lua.

In order to communicate properly with Lua, a C function must follow the following protocol, which defines the way parameters and results are passed: A C function receives its arguments from Lua in its stack in direct order (the first argument is pushed first). So, when the function starts, its first argument (if any) is at index 1. To return values to Lua, a C function just pushes them onto the stack, in direct order (the first result is pushed first), and returns the number of results. Any other value in the stack below the results will be properly discharged by Lua. Like a Lua function, a C function called by Lua can also return many results.

As an example, the following function receives a variable number of numerical arguments and returns their average and sum:

```c
static int foo (lua_State *L) {
  int n = lua_gettop(L);    /* number of arguments */
  lua_Number sum = 0;
  int i;
  for (i = 1; i <= n; i++) {
    if (!lua_isnumber(L, i)) {
      lua_pushstring(L, "incorrect argument to function `aver
      lua_error(L);
    }
    sum += lua_tonumber(L, i);
  }
  lua_pushnumber(L, sum/n);        /* first result */
  lua_pushnumber(L, sum);          /* second result */
  return 2;                        /* number of results */
}
```

To register a C function to Lua, there is the following convenience macro:

```
    #define lua_register(L,n,f) \
            (lua_pushstring(L, n), \
             lua_pushcfunction(L, f), \
             lua_settable(L, LUA_GLOBALSINDEX))
    /* lua_State *L;     */
    /* const char *n;    */
    /* lua_CFunction f; */
```

which receives the name the function will have in Lua and a pointer to the
function. Thus, the C function `foo` above may be registered in Lua as
`average` by calling

```
    lua_register(L, "average", foo);
```

3.17 - Defining C Closures

When a C function is created, it is possible to associate some values with it, thus creating a *C closure*; these values are then accessible to the function whenever it is called. To associate values with a C function, first these values should be pushed onto the stack (when there are multiple values, the first value is pushed first). Then the function

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n)
```

is used to push the C function onto the stack, with the argument `n` telling how many values should be associated with the function (`lua_pushcclosure` also pops these values from the stack); in fact, the macro `lua_pushcfunction` is defined as `lua_pushcclosure` with `n` set to 0.

Then, whenever the C function is called, those values are located at specific pseudo-indices. Those pseudo-indices are produced by a macro `lua_upvalueindex`. The first value associated with a function is at position `lua_upvalueindex(1)`, and so on. Any access to `lua_upvalueindex(`*n*`)`, where *n* is greater than the number of upvalues of the current function, produces an acceptable (but invalid) index.

For examples of C functions and closures, see the standard libraries in the official Lua distribution (`src/lib/*.c`).

3.18 - Registry

Lua provides a registry, a pre-defined table that can be used by any
C code to store whatever Lua value it needs to store, specially if the
C code needs to keep that Lua value outside the life span of a C function.
This table is always located at pseudo-index `LUA_REGISTRYINDEX`. Any
C library can store data into this table, as long as it chooses keys
different from other libraries. Typically, you should use as key a string
containing your library name or a light userdata with the address of a C
object in your code.

The integer keys in the registry are used by the reference mechanism,
implemented by the auxiliary library, and therefore should not be used by
other purposes.

3.19 - Error Handling in C

Internally, Lua uses the C `longjmp` facility to handle errors. When Lua faces any error (such as memory allocation errors, type errors, syntax errors) it *raises* an error, that is, it does a long jump. A *protected environment* uses `setjmp` to set a recover point; any error jumps to the most recent active recover point.

If an error happens outside any protected environment, Lua calls a *panic function* and then calls `exit(EXIT_FAILURE)`. You can change the panic function with

```
lua_CFunction lua_atpanic (lua_State *L, lua_CFunction panicf
```

Your new panic function may avoid the application exit by never returning (e.g., by doing a long jump). Nevertheless, the corresponding Lua state will not be consistent; the only safe operation with it is to close it.

Almost any function in the API may raise an error, for instance due to a memory allocation error. The following functions run in protected mode (that is, they create a protected environment to run), so they never raise an error: `lua_open`, `lua_close`, `lua_load`, and `lua_pcall`.

There is yet another function that runs a given C function in protected mode:

```
int lua_cpcall (lua_State *L, lua_CFunction func, void *ud);
```

`lua_cpcall` calls `func` in protected mode. `func` starts with only one element in its stack, a light userdata containing `ud`. In case of errors, `lua_cpcall` returns the same error codes as `lua_pcall` (see 3.15), plus the error object on the top of the stack; otherwise, it returns zero, and does not change the stack. Any value returned by `func` is discarded.

C code can generate a Lua error calling the function

```
void lua_error (lua_State *L);
```

The error message (which actually can be any type of object) must be on the stack top. This function does a long jump, and therefore never returns.

3.20 - Threads

Lua offers partial support for multiple threads of execution. If you have a C library that offers multi-threading, then Lua can cooperate with it to implement the equivalent facility in Lua. Also, Lua implements its own coroutine system on top of threads. The following function creates a new thread in Lua:

```
lua_State *lua_newthread (lua_State *L);
```

This function pushes the thread on the stack and returns a pointer to a `lua_State` that represents this new thread. The new state returned by this function shares with the original state all global objects (such as tables), but has an independent run-time stack.

Each thread has an independent global environment table. When you create a thread, this table is the same as that of the given state, but you can change each one independently.

There is no explicit function to close or to destroy a thread. Threads are subject to garbage collection, like any Lua object.

To manipulate threads as coroutines, Lua offers the following functions:

```
int lua_resume (lua_State *L, int narg);
int lua_yield  (lua_State *L, int nresults);
```

To start a coroutine, you first create a new thread; then you push on its stack the body function plus any eventual arguments; then you call `lua_resume`, with `narg` being the number of arguments. This call returns when the coroutine suspends or finishes its execution. When it returns, the stack contains all values passed to `lua_yield`, or all values returned by the body function. `lua_resume` returns 0 if there are no errors running the coroutine, or an error code (see 3.15). In case of errors, the stack contains only the error message. To restart a coroutine, you put on its stack only the values to be passed as results from `yield`, and then call `lua_resume`.

The `lua_yield` function can only be called as the return expression of a C function, as follows:

```
return lua_yield (L, nresults);
```

When a C function calls `lua_yield` in that way, the running coroutine suspends its execution, and the call to `lua_resume` that started this coroutine returns. The parameter `nresults` is the number of values from the stack that are passed as results to `lua_resume`.

To exchange values between different threads, you may use `lua_xmove`:

```
void lua_xmove (lua_State *from, lua_State *to, int n);
```

It pops `n` values from the stack `from`, and puhses them into the stack `to`.

4 - The Debug Interface

Lua has no built-in debugging facilities. Instead, it offers a special interface by means of functions and *hooks*. This interface allows the construction of different kinds of debuggers, profilers, and other tools that need "inside information" from the interpreter.

4.1 - Stack and Function Information

The main function to get information about the interpreter runtime stack is

```
int lua_getstack (lua_State *L, int level, lua_Debug *ar);
```

This function fills parts of a `lua_Debug` structure with an identification of the *activation record* of the function executing at a given level. Level 0 is the current running function, whereas level *n+1* is the function that has called level *n*. When there are no errors, `lua_getstack` returns 1; when called with a level greater than the stack depth, it returns 0.

The structure `lua_Debug` is used to carry different pieces of information about an active function:

```
typedef struct lua_Debug {
  int event;
  const char *name;        /* (n) */
  const char *namewhat;    /* (n) `global', `local', `field', `m
  const char *what;        /* (S) `Lua' function, `C' function,
  const char *source;      /* (S) */
  int currentline;         /* (l) */
  int nups;                /* (u) number of upvalues */
  int linedefined;         /* (S) */
  char short_src[LUA_IDSIZE]; /* (S) */

  /* private part */
  ...
} lua_Debug;
```

`lua_getstack` fills only the private part of this structure, for later use. To fill the other fields of `lua_Debug` with useful information, call

```
int lua_getinfo (lua_State *L, const char *what, lua_Debug *a
```

This function returns 0 on error (for instance, an invalid option in `what`). Each character in the string `what` selects some fields of the structure `ar` to be filled, as indicated by the letter in parentheses in the definition of `lua_Debug` above: `s´` fills in the fields `source`, `linedefined`, and `what`; `l´` fills in the field `currentline`, etc. Moreover, `f´` pushes onto the stack the function that is running at the given level.

To get information about a function that is not active (that is, not in the stack), you push it onto the stack and start the `what` string with the character `` ` ``>´. For instance, to know in which line a function `f` was defined, you can write

```
lua_Debug ar;
lua_pushstring(L, "f");
lua_gettable(L, LUA_GLOBALSINDEX);  /* get global `f' */
lua_getinfo(L, ">S", &ar);
printf("%d\n", ar.linedefined);
```

The fields of `lua_Debug` have the following meaning:

- **source** If the function was defined in a string, then `source` is that string. If the function was defined in a file, then `source` starts with a `` ` ``@´ followed by the file name.

- **short_src** A "printable" version of `source`, to be used in error messages.

- **linedefined** the line number where the definition of the function starts.

- **what** the string `"Lua"` if this is a Lua function, `"C"` if this is a C function, `"main"` if this is the main part of a chunk, and `"tail"` if this was a function that did a tail call. In the latter case, Lua has no other information about this function.

- **currentline** the current line where the given function is executing. When no line information is available, `currentline` is set to *-1*.

- **name** a reasonable name for the given function. Because functions in Lua are first class values, they do not have a fixed name: Some functions may be the value of multiple global variables, while others may be stored only in a table field. The `lua_getinfo` function checks how the function was called or whether it is the value of a global variable to find a suitable name. If it cannot find a name, then `name` is set to `NULL`.

- **namewhat** Explains the `name` field. The value of `namewhat` can be

"global", "local", "method", "field", or "" (the empty string), according to how the function was called. (Lua uses the empty string when no other option seems to apply.)

- **nups** The number of upvalues of the function.

4.2 - Manipulating Local Variables and Upvalues

For the manipulation of local variables and upvalues, the debug interface uses indices: The first parameter or local variable has index 1, and so on, until the last active local variable. Upvalues have no particular order, as they are active through the whole function.

The following functions allow the manipulation of the local variables of a given activation record:

```
const char *lua_getlocal (lua_State *L, const lua_Debug *ar,
const char *lua_setlocal (lua_State *L, const lua_Debug *ar,
```

The parameter `ar` must be a valid activation record that was filled by a previous call to `lua_getstack` or given as argument to a hook (see [4.3](#)). `lua_getlocal` gets the index `n` of a local variable, pushes the variable's value onto the stack, and returns its name. `lua_setlocal` assigns the value at the top of the stack to the variable and returns its name. Both functions return `NULL` when the index is greater than the number of active local variables.

The following functions allow the manipulation of the upvalues of a given function (unlike local variables, the upvalues of a function are accessible even when the function is not active):

```
const char *lua_getupvalue (lua_State *L, int funcindex, int
const char *lua_setupvalue (lua_State *L, int funcindex, int
```

These functions operate both on Lua functions and on C functions. (For Lua functions, upvalues are the external local variables that the function uses, and that consequently are included in its closure.) `funcindex` points to a function in the stack. `lua_getupvalue` gets the index `n` of an upvalue, pushes the upvalue's value onto the stack, and returns its name. `lua_setupvalue` assigns the value at the top of the stack to the upvalue and returns its name. Both functions return `NULL` when the index is greater than the number of upvalues. For C functions, these functions use the empty string `""` as a name for all upvalues.

As an example, the following function lists the names of all local variables and upvalues for a function at a given level of the stack:

```c
int listvars (lua_State *L, int level) {
  lua_Debug ar;
  int i;
  const char *name;
  if (lua_getstack(L, level, &ar) == 0)
    return 0;  /* failure: no such level in the stack */
  i = 1;
  while ((name = lua_getlocal(L, &ar, i++)) != NULL) {
    printf("local %d %s\n", i-1, name);
    lua_pop(L, 1);  /* remove variable value */
  }
  lua_getinfo(L, "f", &ar);  /* retrieves function */
  i = 1;
  while ((name = lua_getupvalue(L, -1, i++)) != NULL) {
    printf("upvalue %d %s\n", i-1, name);
    lua_pop(L, 1);  /* remove upvalue value */
  }
  return 1;
}
```

4.3 - Hooks

Lua offers a mechanism of hooks, which are user-defined C functions that are called during the program execution. A hook may be called in four different events: a *call* event, when Lua calls a function; a *return* event, when Lua returns from a function; a *line* event, when Lua starts executing a new line of code; and a *count* event, which happens every "count" instructions. Lua identifies these events with the following constants: `LUA_HOOKCALL`, `LUA_HOOKRET` (or `LUA_HOOKTAILRET`, see below), `LUA_HOOKLINE`, and `LUA_HOOKCOUNT`.

A hook has type `lua_Hook`, defined as follows:

```
typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);
```

You can set the hook with the following function:

```
int lua_sethook (lua_State *L, lua_Hook func, int mask, int c
```

`func` is the hook. `mask` specifies on which events the hook will be called: It is formed by a disjunction of the constants `LUA_MASKCALL`, `LUA_MASKRET`, `LUA_MASKLINE`, and `LUA_MASKCOUNT`. The `count` argument is only meaningful when the mask includes `LUA_MASKCOUNT`. For each event, the hook is called as explained below:

- **The call hook** is called when the interpreter calls a function. The hook is called just after Lua enters the new function.
- **The return hook** is called when the interpreter returns from a function. The hook is called just before Lua leaves the function.
- **The line hook** is called when the interpreter is about to start the execution of a new line of code, or when it jumps back in the code (even to the same line). (This event only happens while Lua is executing a Lua function.)
- **The count hook** is called after the interpreter executes every `count` instructions. (This event only happens while Lua is executing a Lua function.)

A hook is disabled by setting `mask` to zero.

You can get the current hook, the current mask, and the current count with the following functions:

```
lua_Hook lua_gethook      (lua_State *L);
int      lua_gethookmask  (lua_State *L);
int      lua_gethookcount (lua_State *L);
```

Whenever a hook is called, its `ar` argument has its field `event` set to the specific event that triggered the hook. Moreover, for line events, the field `currentline` is also set. To get the value of any other field in `ar`, the hook must call `lua_getinfo`. For return events, `event` may be `LUA_HOOKRET`, the normal value, or `LUA_HOOKTAILRET`. In the latter case, Lua is simulating a return from a function that did a tail call; in this case, it is useless to call `lua_getinfo`.

While Lua is running a hook, it disables other calls to hooks. Therefore, if a hook calls back Lua to execute a function or a chunk, that execution occurs without any calls to hooks.

5 - Standard Libraries

The standard libraries provide useful functions that are implemented directly through the C API. Some of these functions provide essential services to the language (e.g., `type` and `getmetatable`); others provide access to "outside" services (e.g., I/O); and others could be implemented in Lua itself, but are quite useful or have critical performance to deserve an implementation in C (e.g., `sort`).

All libraries are implemented through the official C API and are provided as separate C modules. Currently, Lua has the following standard libraries:

- basic library;
- string manipulation;
- table manipulation;
- mathematical functions (sin, log, etc.);
- input and output;
- operating system facilities;
- debug facilities.

Except for the basic library, each library provides all its functions as fields of a global table or as methods of its objects.

To have access to these libraries, the C host program must first call the functions `luaopen_base` (for the basic library), `luaopen_string` (for the string library), `luaopen_table` (for the table library), `luaopen_math` (for the mathematical library), `luaopen_io` (for the I/O and the Operating System libraries), and `luaopen_debug` (for the debug library). These functions are declared in `lualib.h`.

5.1 - Basic Functions

The basic library provides some core functions to Lua. If you do not include this library in your application, you should check carefully whether you need to provide some alternative implementation for some of its facilities.

assert (v [, message])

Issues an error when the value of its argument `v` is **nil** or **false**; otherwise, returns this value. `message` is an error message; when absent, it defaults to "assertion failed!"

collectgarbage ([limit])

Sets the garbage-collection threshold to the given limit (in Kbytes) and checks it against the byte counter. If the new threshold is smaller than the byte counter, then Lua immediately runs the garbage collector (see [2.9](#)). If `limit` is absent, it defaults to zero (thus forcing a garbage-collection cycle).

dofile (filename)

Opens the named file and executes its contents as a Lua chunk. When called without arguments, `dofile` executes the contents of the standard input (`stdin`). Returns any value returned by the chunk. In case of errors, `dofile` propagates the error to its caller (that is, it does not run in protected mode).

error (message [, level])

Terminates the last protected function called and returns `message` as the error message. Function `error` never returns.

The `level` argument specifies where the error message points the error. With level 1 (the default), the error position is where the `error` function was called. Level 2 points the error to where the function that called `error`

was called; and so on.

_G

A global variable (not a function) that holds the global environment (that is, `_G._G = _G`). Lua itself does not use this variable; changing its value does not affect any environment. (Use `setfenv` to change environments.)

getfenv (f)

Returns the current environment in use by the function. `f` can be a Lua function or a number, which specifies the function at that stack level: Level 1 is the function calling `getfenv`. If the given function is not a Lua function, or if `f` is 0, `getfenv` returns the global environment. The default for `f` is 1.

If the environment has a "`__fenv`" field, returns the associated value, instead of the environment.

getmetatable (object)

If the object does not have a metatable, returns **nil**. Otherwise, if the object's metatable has a "`__metatable`" field, returns the associated value. Otherwise, returns the metatable of the given object.

gcinfo ()

Returns two results: the number of Kbytes of dynamic memory that Lua is using and the current garbage collector threshold (also in Kbytes).

ipairs (t)

Returns an iterator function, the table `t`, and 0, so that the construction

```
for i,v in ipairs(t) do ... end
```

will iterate over the pairs (`1`,`t[1]`), (`2`,`t[2]`), ..., up to the first integer key with a nil value in the table.

loadfile (filename)

Loads a file as a Lua chunk (without running it). If there are no errors, returns the compiled chunk as a function; otherwise, returns **nil** plus the error message. The environment of the returned function is the global environment.

loadlib (libname, funcname)

Links the program with the dynamic C library `libname`. Inside this library, looks for a function `funcname` and returns this function as a C function.

`libname` must be the complete file name of the C library, including any eventual path and extension.

This function is not supported by ANSI C. As such, it is only available on some platforms (Windows, Linux, Solaris, BSD, plus other Unix systems that support the `dlfcn` standard).

loadstring (string [, chunkname])

Loads a string as a Lua chunk (without running it). If there are no errors, returns the compiled chunk as a function; otherwise, returns **nil** plus the error message. The environment of the returned function is the global environment.

The optional parameter `chunkname` is the name to be used in error messages and debug information.

To load and run a given string, use the idiom

```
assert(loadstring(s))()
```

next (table [, index])

Allows a program to traverse all fields of a table. Its first argument is a table and its second argument is an index in this table. `next` returns the next index of the table and the value associated with the index. When called with **nil** as its second argument, `next` returns the first index of the

table and its associated value. When called with the last index, or with **nil** in an empty table, `next` returns **nil**. If the second argument is absent, then it is interpreted as **nil**.

Lua has no declaration of fields; There is no difference between a field not present in a table or a field with value **nil**. Therefore, `next` only considers fields with non-**nil** values. The order in which the indices are enumerated is not specified, *even for numeric indices*. (To traverse a table in numeric order, use a numerical **for** or the `ipairs` function.)

The behavior of `next` is *undefined* if, during the traversal, you assign any value to a non-existent field in the table.

pairs (t)

Returns the `next` function and the table `t` (plus a **nil**), so that the construction

```
        for k,v in pairs(t) do ... end
```

will iterate over all key-value pairs of table `t`.

pcall (f, arg1, arg2, ...)

Calls function `f` with the given arguments in protected mode. That means that any error inside `f` is not propagated; instead, `pcall` catches the error and returns a status code. Its first result is the status code (a boolean), which is **true** if the call succeeds without errors. In such case, `pcall` also returns all results from the call, after this first result. In case of any error, `pcall` returns **false** plus the error message.

print (e1, e2, ...)

Receives any number of arguments, and prints their values in `stdout`, using the `tostring` function to convert them to strings. This function is not intended for formatted output, but only as a quick way to show a value, typically for debugging. For formatted output, use `format` (see 5.3).

rawequal (v1, v2)

Checks whether `v1` is equal to `v2`, without invoking any metamethod. Returns a boolean.

`rawget (table, index)`

Gets the real value of `table[index]`, without invoking any metamethod. `table` must be a table; `index` is any value different from **nil**.

`rawset (table, index, value)`

Sets the real value of `table[index]` to `value`, without invoking any metamethod. `table` must be a table, `index` is any value different from **nil**, and `value` is any Lua value.

`require (packagename)`

Loads the given package. The function starts by looking into the table `_LOADED` to determine whether `packagename` is already loaded. If it is, then `require` returns the value that the package returned when it was first loaded. Otherwise, it searches a path looking for a file to load.

If the global variable `LUA_PATH` is a string, this string is the path. Otherwise, `require` tries the environment variable `LUA_PATH`. As a last resort, it uses the predefined path `"?;?.lua"`.

The path is a sequence of *templates* separated by semicolons. For each template, `require` will change each interrogation mark in the template to `packagename`, and then will try to load the resulting file name. So, for instance, if the path is

```
"./?.lua;./?.lc;/usr/local/?/?.lua;/lasttry"
```

a `require "mod"` will try to load the files `./mod.lua`, `./mod.lc`, `/usr/local/mod/mod.lua`, and `/lasttry`, in that order.

The function stops the search as soon as it can load a file, and then it runs the file. After that, it associates, in table `_LOADED`, the package name with the value that the package returned, and returns that value. If the package returns **nil** (or no value), `require` converts this value to **true**. If

the package returns **false**, `require` also returns **false**. However, as the mark in table `_LOADED` is **false**, any new attempt to reload the file will happen as if the package was not loaded (that is, the package will be loaded again).

If there is any error loading or running the file, or if it cannot find any file in the path, then `require` signals an error.

While running a file, `require` defines the global variable `_REQUIREDNAME` with the package name. The package being loaded always runs within the global environment.

setfenv (f, table)

Sets the current environment to be used by the given function. `f` can be a Lua function or a number, which specifies the function at that stack level: Level 1 is the function calling `setfenv`.

As a special case, when `f` is 0 `setfenv` changes the global environment of the running thread.

If the original environment has a `"__fenv"` field, `setfenv` raises an error.

setmetatable (table, metatable)

Sets the metatable for the given table. (You cannot change the metatable of a userdata from Lua.) If `metatable` is **nil**, removes the metatable of the given table. If the original metatable has a `"__metatable"` field, raises an error.

tonumber (e [, base])

Tries to convert its argument to a number. If the argument is already a number or a string convertible to a number, then `tonumber` returns that number; otherwise, it returns **nil**.

An optional argument specifies the base to interpret the numeral. The base may be any integer between 2 and 36, inclusive. In bases above 10, the letter `A` (in either upper or lower case) represents 10, `B`

represents 11, and so forth, with `z´ representing 35. In base 10 (the default), the number may have a decimal part, as well as an optional exponent part (see 2.2.1). In other bases, only unsigned integers are accepted.

tostring (e)

Receives an argument of any type and converts it to a string in a reasonable format. For complete control of how numbers are converted, use `format` (see 5.3).

If the metatable of `e` has a `"__tostring"` field, `tostring` calls the corresponding value with `e` as argument, and uses the result of the call as its result.

type (v)

Returns the type of its only argument, coded as a string. The possible results of this function are `"nil"` (a string, not the value **nil**), `"number"`, `"string"`, `"boolean`, `"table"`, `"function"`, `"thread"`, and `"userdata"`.

unpack (list)

Returns all elements from the given list. This function is equivalent to

```
return list[1], list[2], ..., list[n]
```

except that the above code can be written only for a fixed *n*. The number *n* is the size of the list, as defined for the `table.getn` function.

_VERSION

A global variable (not a function) that holds a string containing the current interpreter version. The current content of this string is `"Lua 5.0"`.

xpcall (f, err)

This function is similar to `pcall`, except that you can set a new error handler.

`xpcall` calls function `f` in protected mode, using `err` as the error handler. Any error inside `f` is not propagated; instead, `xpcall` catches the error, calls the `err` function with the original error object, and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In such case, `xpcall` also returns all results from the call, after this first result. In case of any error, `xpcall` returns false plus the result from `err`.

5.2 - Coroutine Manipulation

The operations related to coroutines comprise a sub-library of the basic library and come inside the table `coroutine`. See [2.10](#) for a general description of coroutines.

coroutine.create (f)

Creates a new coroutine, with body `f`. `f` must be a Lua function. Returns this new coroutine, an object with type `"thread"`.

coroutine.resume (co, val1, ...)

Starts or continues the execution of coroutine `co`. The first time you resume a coroutine, it starts running its body. The arguments `val1`, ... go as the arguments to the body function. If the coroutine has yielded, `resume` restarts it; the arguments `val1`, ... go as the results from the yield.

If the coroutine runs without any errors, `resume` returns **true** plus any values passed to `yield` (if the coroutine yields) or any values returned by the body function (if the coroutine terminates). If there is any error, `resume` returns **false** plus the error message.

coroutine.status (co)

Returns the status of coroutine `co`, as a string: `"running"`, if the coroutine is running (that is, it called `status`); `"suspended"`, if the coroutine is suspended in a call to `yield`, or if it has not started running yet; and `"dead"` if the coroutine has finished its body function, or if it has stopped with an error.

coroutine.wrap (f)

Creates a new coroutine, with body `f`. `f` must be a Lua function. Returns a function that resumes the coroutine each time it is called. Any arguments passed to the function behave as the extra arguments to `resume`. Returns the same values returned by `resume`, except the first

boolean. In case of error, propagates the error.

coroutine.yield (val1, ...)

Suspends the execution of the calling coroutine. The coroutine cannot be running neither a C function, nor a metamethod, nor an iterator. Any arguments to `yield` go as extra results to `resume`.

5.3 - String Manipulation

This library provides generic functions for string manipulation, such as finding and extracting substrings, and pattern matching. When indexing a string in Lua, the first character is at position 1 (not at 0, as in C). Indices are allowed to be negative and are interpreted as indexing backwards, from the end of the string. Thus, the last character is at position *-1*, and so on.

The string library provides all its functions inside the table `string`.

string.byte (s [, i])

Returns the internal numerical code of the `i`-th character of `s`, or **nil** if the index is out of range. If `i` is absent, then it is assumed to be 1. `i` may be negative.

Note that numerical codes are not necessarily portable across platforms.

string.char (i1, i2, ...)

Receives 0 or more integers. Returns a string with length equal to the number of arguments, in which each character has the internal numerical code equal to its correspondent argument.

Note that numerical codes are not necessarily portable across platforms.

string.dump (function)

Returns a binary representation of the given function, so that a later `loadstring` on that string returns a copy of the function. `function` must be a Lua function without upvalues.

string.find (s, pattern [, init [, plain]])

Looks for the first *match* of `pattern` in the string `s`. If it finds one, then `find` returns the indices of `s` where this occurrence starts and ends;

otherwise, it returns **nil**. If the pattern specifies captures (see `string.gsub` below), the captured strings are returned as extra results. A third, optional numerical argument `init` specifies where to start the search; it may be negative and its default value is 1. A value of **true** as a fourth, optional argument `plain` turns off the pattern matching facilities, so the function does a plain "find substring" operation, with no characters in `pattern` being considered "magic". Note that if `plain` is given, then `init` must be given too.

string.len (s)

Receives a string and returns its length. The empty string "" has length 0. Embedded zeros are counted, so `"a\000b\000c"` has length 5.

string.lower (s)

Receives a string and returns a copy of that string with all uppercase letters changed to lowercase. All other characters are left unchanged. The definition of what is an uppercase letter depends on the current locale.

string.rep (s, n)

Returns a string that is the concatenation of `n` copies of the string `s`.

string.sub (s, i [, j])

Returns the substring of `s` that starts at `i` and continues until `j`; `i` and `j` may be negative. If `j` is absent, then it is assumed to be equal to *-1* (which is the same as the string length). In particular, the call `string.sub(s,1,j)` returns a prefix of `s` with length `j`, and `string.sub(s, -i)` returns a suffix of `s` with length `i`.

string.upper (s)

Receives a string and returns a copy of that string with all lowercase letters changed to uppercase. All other characters are left unchanged. The definition of what is a lowercase letter depends on the current locale.

string.format (formatstring, e1, e2, ...)

Returns a formatted version of its variable number of arguments following the description given in its first argument (which must be a string). The format string follows the same rules as the `printf` family of standard C functions. The only differences are that the options/modifiers `*`, `l`, `L`, `n`, `p`, and `h` are not supported, and there is an extra option, `q`. The `q` option formats a string in a form suitable to be safely read back by the Lua interpreter: The string is written between double quotes, and all double quotes, newlines, and backslashes in the string are correctly escaped when written. For instance, the call

```
string.format('%q', 'a string with "quotes" and \n new line')
```

will produce the string:

```
"a string with \"quotes\" and \
 new line"
```

The options `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X`, and `x` all expect a number as argument, whereas `q` and `s` expect a string. The `*` modifier can be simulated by building the appropriate format string. For example, `"%*g"` can be simulated with `"%"..width.."g"`.

String values to be formatted with `%s` cannot contain embedded zeros.

string.gfind (s, pat)

Returns an iterator function that, each time it is called, returns the next captures from pattern `pat` over string `s`.

If `pat` specifies no captures, then the whole match is produced in each call.

As an example, the following loop

```
s = "hello world from Lua"
for w in string.gfind(s, "%a+") do
  print(w)
end
```

will iterate over all the words from string `s`, printing one per line. The next example collects all pairs `key=value` from the given string into a table:

```
t = {}
s = "from=world, to=Lua"
for k, v in string.gfind(s, "(%w+)=(%w+)") do
  t[k] = v
end
```

string.gsub (s, pat, repl [, n])

Returns a copy of `s` in which all occurrences of the pattern `pat` have been replaced by a replacement string specified by `repl`. `gsub` also returns, as a second value, the total number of substitutions made.

If `repl` is a string, then its value is used for replacement. Any sequence in `repl` of the form %*n*, with *n* between 1 and 9, stands for the value of the *n*-th captured substring (see below).

If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order; if the pattern specifies no captures, then the whole match is passed as a sole argument. If the value returned by this function is a string, then it is used as the replacement string; otherwise, the replacement string is the empty string.

The optional last parameter `n` limits the maximum number of substitutions to occur. For instance, when `n` is 1 only the first occurrence of `pat` is replaced.

Here are some examples:

```
x = string.gsub("hello world", "(%w+)", "%1 %1")
--> x="hello hello world world"

x = string.gsub("hello world", "(%w+)", "%1 %1", 1)
--> x="hello hello world"

x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
--> x="world hello Lua from"

x = string.gsub("home = $HOME, user = $USER", "%$(%w+)", os.geten
```

```
--> x="home = /home/roberto, user = roberto"

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
      return loadstring(s)()
    end)
--> x="4+5 = 9"

local t = {name="lua", version="5.0"}
x = string.gsub("$name_$version.tar.gz", "%$(%w+)", function (v)
      return t[v]
    end)
--> x="lua_5.0.tar.gz"
```

Patterns

A *character class* is used to represent a set of characters. The following
combinations are allowed in describing a character class:

- **x** (where *x* is not one of the magic characters `^$()%.[]*+-?`) ---
  represents the character *x* itself.
- **.** --- (a dot) represents all characters.
- **%a** --- represents all letters.
- **%c** --- represents all control characters.
- **%d** --- represents all digits.
- **%l** --- represents all lowercase letters.
- **%p** --- represents all punctuation characters.
- **%s** --- represents all space characters.
- **%u** --- represents all uppercase letters.
- **%w** --- represents all alphanumeric characters.
- **%x** --- represents all hexadecimal digits.
- **%z** --- represents the character with representation 0.
- **%***x* (where *x* is any non-alphanumeric character) --- represents the
  character *x*. This is the standard way to escape the magic
  characters. Any punctuation character (even the non magic) can be
  preceded by a `%´ when used to represent itself in a pattern.

- **[*set*]** --- represents the class which is the union of all characters in
  set. A range of characters may be specified by separating the end
  characters of the range with a `-´. All classes %*x* described above
  may also be used as components in *set*. All other characters in *set*
```

represent themselves. For example, `[%w_]` (or `[_%w]`) represents all alphanumeric characters plus the underscore, `[0-7]` represents the octal digits, and `[0-7%l%-]` represents the octal digits plus the lowercase letters plus the `` `-´ `` character.

The interaction between ranges and classes is not defined. Therefore, patterns like `[%a-z]` or `[a-%%]` have no meaning.

- **[^*set*]** --- represents the complement of *set*, where *set* is interpreted as above.

For all classes represented by single letters (`%a`, `%c`, etc.), the corresponding uppercase letter represents the complement of the class. For instance, `%S` represents all non-space characters.

The definitions of letter, space, and other character groups depend on the current locale. In particular, the class `[a-z]` may not be equivalent to `%l`. The second form should be preferred for portability.

A *pattern item* may be

- a single character class, which matches any single character in the class;
- a single character class followed by `` `*´ ``, which matches 0 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by `` `+´ ``, which matches 1 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by `` `-´ ``, which also matches 0 or more repetitions of characters in the class. Unlike `` `*´ ``, these repetition items will always match the *shortest* possible sequence;
- a single character class followed by `` `?´ ``, which matches 0 or 1 occurrence of a character in the class;
- `%n`, for *n* between 1 and 9; such item matches a substring equal to the *n*-th captured string (see below);
- `%bxy`, where *x* and *y* are two distinct characters; such item matches strings that start with *x*, end with *y*, and where the *x* and *y* are

*balanced*. This means that, if one reads the string from left to right, counting *+1* for an *x* and *-1* for a *y*, the ending *y* is the first *y* where the count reaches 0. For instance, the item `%b()` matches expressions with balanced parentheses.

A *pattern* is a sequence of pattern items. A `` `^´ `` at the beginning of a pattern anchors the match at the beginning of the subject string. A `` `$´ `` at the end of a pattern anchors the match at the end of the subject string. At other positions, `` `^´ `` and `` `$´ `` have no special meaning and represent themselves.

A pattern may contain sub-patterns enclosed in parentheses; they describe *captures*. When a match succeeds, the substrings of the subject string that match captures are stored (*captured*) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern `"(a*(.)%w(%s*))"`, the part of the string matching `"a*(.)%w(%s*)"` is stored as the first capture (and therefore has number 1); the character matching `.` is captured with number 2, and the part matching `%s*` has number 3.

As a special case, the empty capture `()` captures the current string position (a number). For instance, if we apply the pattern `"()aa()"` on the string `"flaaap"`, there will be two captures: 3 and 5.

A pattern cannot contain embedded zeros. Use `%z` instead.

# 5.4 - Table Manipulation

This library provides generic functions for table manipulation. It provides all its functions inside the table `table`.

Most functions in the table library assume that the table represents an array or a list. For those functions, an important concept is the *size* of the array. There are three ways to specify that size:

- the field `"n"` --- When the table has a field `"n"` with a numerical value, that value is assumed as its size.
- `setn` --- You can call the `table.setn` function to explicitly set the size of a table.
- implicit size --- Otherwise, the size of the object is one less the first integer index with a **nil** value.

For more details, see the descriptions of the `table.getn` and `table.setn` functions.

**table.concat (table [, sep [, i [, j]]])**

Returns `table[i]..sep..table[i+1] ... sep..table[j]`. The default value for `sep` is the empty string, the default for `i` is 1, and the default for `j` is the size of the table. If `i` is greater than `j`, returns the empty string.

**table.foreach (table, f)**

Executes the given `f` over all elements of `table`. For each element, `f` is called with the index and respective value as arguments. If `f` returns a non-**nil** value, then the loop is broken, and this value is returned as the final value of `foreach`.

See the `next` function for extra information about table traversals.

**table.foreachi (table, f)**

Executes the given `f` over the numerical indices of `table`. For each index,

`f` is called with the index and respective value as arguments. Indices are visited in sequential order, from 1 to `n`, where `n` is the size of the table (see 5.4). If `f` returns a non-**nil** value, then the loop is broken and this value is returned as the result of `foreachi`.

**`table.getn (table)`**

Returns the size of a table, when seen as a list. If the table has an `n` field with a numeric value, this value is the size of the table. Otherwise, if there was a previous call to `table.setn` over this table, the respective value is returned. Otherwise, the size is one less the first integer index with a **nil** value.

**`table.sort (table [, comp])`**

Sorts table elements in a given order, *in-place*, from `table[1]` to `table[n]`, where `n` is the size of the table (see 5.4). If `comp` is given, then it must be a function that receives two table elements, and returns true when the first is less than the second (so that `not comp(a[i+1],a[i])` will be true after the sort). If `comp` is not given, then the standard Lua operator `<` is used instead.

The sort algorithm is *not* stable, that is, elements considered equal by the given order may have their relative positions changed by the sort.

**`table.insert (table, [pos,] value)`**

Inserts element `value` at position `pos` in `table`, shifting up other elements to open space, if necessary. The default value for `pos` is `n+1`, where `n` is the size of the table (see 5.4), so that a call `table.insert(t,x)` inserts `x` at the end of table `t`. This function also updates the size of the table by calling `table.setn(table, n+1)`.

**`table.remove (table [, pos])`**

Removes from `table` the element at position `pos`, shifting down other elements to close the space, if necessary. Returns the value of the removed element. The default value for `pos` is `n`, where `n` is the size of the

table (see [5.4](#)), so that a call `table.remove(t)` removes the last element of table `t`. This function also updates the size of the table by calling `table.setn(table, n-1)`.

**table.setn (table, n)**

Updates the size of a table. If the table has a field `"n"` with a numerical value, that value is changed to the given `n`. Otherwise, it updates an internal state so that subsequent calls to `table.getn(table)` return `n`.

# 5.5 - Mathematical Functions

This library is an interface to most of the functions of the standard C math library. (Some have slightly different names.) It provides all its functions inside the table `math`. In addition, it registers the global `__pow` for the binary exponentiation operator `^`, so that `x^y` returns $x^y$. The library provides the following functions:

```
math.abs     math.acos    math.asin    math.atan    math.atan
math.ceil    math.cos     math.deg     math.exp     math.floo
math.log     math.log10   math.max     math.min     math.mod
math.pow     math.rad     math.sin     math.sqrt    math.tan
math.frexp   math.ldexp   math.random  math.randomseed
```

plus a variable `math.pi`. Most of them are only interfaces to the corresponding functions in the C library. All trigonometric functions work in radians (previous versions of Lua used degrees). The functions `math.deg` and `math.rad` convert between radians and degrees.

The function `math.max` returns the maximum value of its numeric arguments. Similarly, `math.min` computes the minimum. Both can be used with 1, 2, or more arguments.

The functions `math.random` and `math.randomseed` are interfaces to the simple random generator functions `rand` and `srand` that are provided by ANSI C. (No guarantees can be given for their statistical properties.) When called without arguments, `math.random` returns a pseudo-random real number in the range *[0,1)*. When called with a number *n*, `math.random` returns a pseudo-random integer in the range *[1,n]*. When called with two arguments, *l* and *u*, `math.random` returns a pseudo-random integer in the range *[l,u]*. The `math.randomseed` function sets a "seed" for the pseudo-random generator: Equal seeds produce equal sequences of numbers.

5.6 - Input and Output Facilities

The I/O library provides two different styles for file manipulation. The first one uses implicit file descriptors, that is, there are operations to set a default input file and a default output file, and all input/output operations are over those default files. The second style uses explicit file descriptors.

When using implicit file descriptors, all operations are supplied by table `io`. When using explicit file descriptors, the operation `io.open` returns a file descriptor and then all operations are supplied as methods by the file descriptor.

The table `io` also provides three predefined file descriptors with their usual meanings from C: `io.stdin`, `io.stdout`, and `io.stderr`.

A file handle is a userdata containing the file stream (`FILE*`), with a distinctive metatable created by the I/O library.

Unless otherwise stated, all I/O functions return **nil** on failure (plus an error message as a second result) and some value different from **nil** on success.

io.close ([file])

Equivalent to `file:close`. Without a `file`, closes the default output file.

io.flush ()

Equivalent to `file:flush` over the default output file.

io.input ([file])

When called with a file name, it opens the named file (in text mode), and sets its handle as the default input file. When called with a file handle, it simply sets that file handle as the default input file. When called without parameters, it returns the current default input file.

In case of errors this function raises the error, instead of returning an error code.

io.lines ([filename])

Opens the given file name in read mode and returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

```
for line in io.lines(filename) do ... end
```

will iterate over all lines of the file. When the iterator function detects the end of file, it returns **nil** (to finish the loop) and automatically closes the file.

The call `io.lines()` (without a file name) is equivalent to `io.input():lines()`, that is, it iterates over the lines of the default input file.

io.open (filename [, mode])

This function opens a file, in the mode specified in the string `mode`. It returns a new file handle, or, in case of errors, **nil** plus an error message.

The `mode` string can be any of the following:

- **"r"** read mode (the default);
- **"w"** write mode;
- **"a"** append mode;
- **"r+"** update mode, all previous data is preserved;
- **"w+"** update mode, all previous data is erased;
- **"a+"** append update mode, previous data is preserved, writing is only allowed at the end of file.

The `mode` string may also have a `b` at the end, which is needed in some systems to open the file in binary mode. This string is exactly what is used in the standard C function `fopen`.

io.output ([file])

Similar to `io.input`, but operates over the default output file.

io.read (format1, ...)

Equivalent to `io.input():read`.

io.tmpfile ()

Returns a handle for a temporary file. This file is open in update mode and it is automatically removed when the program ends.

io.type (obj)

Checks whether `obj` is a valid file handle. Returns the string `"file"` if `obj` is an open file handle, `"closed file"` if `obj` is a closed file handle, and **nil** if `obj` is not a file handle.

io.write (value1, ...)

Equivalent to `io.output():write`.

file:close ()

Closes `file`.

file:flush ()

Saves any written data to `file`.

file:lines ()

Returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

```
      for line in file:lines() do ... end
```

will iterate over all lines of the file. (Unlike `io.lines`, this function does not close the file when the loop ends.)

file:read (format1, ...)

Reads the file `file`, according to the given formats, which specify what to read. For each format, the function returns a string (or a number) with the characters read, or **nil** if it cannot read data with the specified format. When called without formats, it uses a default format that reads the entire next line (see below).

The available formats are

- **"*n"** reads a number; this is the only format that returns a number instead of a string.
- **"*a"** reads the whole file, starting at the current position. On end of file, it returns the empty string.
- **"*l"** reads the next line (skipping the end of line), returning **nil** on end of file. This is the default format.
- *number* reads a string with up to that number of characters, returning **nil** on end of file. If number is zero, it reads nothing and returns an empty string, or **nil** on end of file.

file:seek ([whence] [, offset])

Sets and gets the file position, measured from the beginning of the file, to the position given by `offset` plus a base specified by the string `whence`, as follows:

- **"set"** base is position 0 (beginning of the file);
- **"cur"** base is current position;
- **"end"** base is end of file;

In case of success, function `seek` returns the final file position, measured in bytes from the beginning of the file. If this function fails, it returns **nil**, plus a string describing the error.

The default value for `whence` is "cur", and for `offset` is 0. Therefore, the call `file:seek()` returns the current file position, without changing it; the call `file:seek("set")` sets the position to the beginning of the file (and returns 0); and the call `file:seek("end")` sets the position to the end of the file, and returns its size.

file:write (value1, ...)

Writes the value of each of its arguments to the filehandle `file`. The arguments must be strings or numbers. To write other values, use `tostring` or `string.format` before `write`.

5.7 - Operating System Facilities

This library is implemented through table `os`.

os.clock ()

Returns an approximation of the amount of CPU time used by the program, in seconds.

os.date ([format [, time]])

Returns a string or a table containing date and time, formatted according to the given string `format`.

If the `time` argument is present, this is the time to be formatted (see the `os.time` function for a description of this value). Otherwise, `date` formats the current time.

If `format` starts with `` `!´ ``, then the date is formatted in Coordinated Universal Time. After that optional character, if `format` is `*t`, then `date` returns a table with the following fields: `year` (four digits), `month` (1--12), `day` (1--31), `hour` (0--23), `min` (0--59), `sec` (0--61), `wday` (weekday, Sunday is 1), `yday` (day of the year), and `isdst` (daylight saving flag, a boolean).

If `format` is not `*t`, then `date` returns the date as a string, formatted according to the same rules as the C function `strftime`.

When called without arguments, `date` returns a reasonable date and time representation that depends on the host system and on the current locale (that is, `os.date()` is equivalent to `os.date("%c")`).

os.difftime (t2, t1)

Returns the number of seconds from time `t1` to time `t2`. In Posix, Windows, and some other systems, this value is exactly `t2-t1`.

os.execute (command)

This function is equivalent to the C function `system`. It passes `command` to be executed by an operating system shell. It returns a status code, which is system-dependent.

os.exit ([code])

Calls the C function `exit`, with an optional `code`, to terminate the host program. The default value for `code` is the success code.

os.getenv (varname)

Returns the value of the process environment variable `varname`, or **nil** if the variable is not defined.

os.remove (filename)

Deletes the file with the given name. If this function fails, it returns **nil**, plus a string describing the error.

os.rename (oldname, newname)

Renames file named `oldname` to `newname`. If this function fails, it returns **nil**, plus a string describing the error.

os.setlocale (locale [, category])

Sets the current locale of the program. `locale` is a string specifying a locale; `category` is an optional string describing which category to change: `"all"`, `"collate"`, `"ctype"`, `"monetary"`, `"numeric"`, or `"time"`; the default category is `"all"`. The function returns the name of the new locale, or **nil** if the request cannot be honored.

os.time ([table])

Returns the current time when called without arguments, or a time representing the date and time specified by the given table. This table must have fields `year`, `month`, and `day`, and may have fields `hour`, `min`, `sec`, and `isdst` (for a description of these fields, see the `os.date` function).

The returned value is a number, whose meaning depends on your system. In Posix, Windows, and some other systems, this number counts the number of seconds since some given start time (the "epoch"). In other systems, the meaning is not specified, and the number returned by `time` can be used only as an argument to `date` and `difftime`.

os.tmpname ()

Returns a string with a file name that can be used for a temporary file. The file must be explicitly opened before its use and removed when no longer needed.

This function is equivalent to the `tmpnam` C function, and many people (and even some compilers!) advise against its use, because between the time you call this function and the time you open the file, it is possible for another process to create a file with the same name.

5.8 - The Reflexive Debug Interface

The `debug` library provides the functionality of the debug interface to Lua programs. You should exert care when using this library. The functions provided here should be used exclusively for debugging and similar tasks, such as profiling. Please resist the temptation to use them as a usual programming tool: They can be very slow. Moreover, `setlocal` and `getlocal` violate the privacy of local variables and therefore can compromise some otherwise secure code.

All functions in this library are provided inside a `debug` table.

debug.debug ()

Enters an interactive mode with the user, running each string that the user enters. Using simple commands and other debug facilities, the user can inspect global and local variables, change their values, evaluate expressions, and so on. A line containing only the word `cont` finishes this function, so that the caller continues its execution.

Note that commands for `debug.debug` are not lexically nested with any function, so they have no direct access to local variables.

debug.gethook ()

Returns the current hook settings, as three values: the current hook function, the current hook mask, and the current hook count (as set by the `debug.sethook` function).

debug.getinfo (function [, what])

This function returns a table with information about a function. You can give the function directly, or you can give a number as the value of `function`, which means the function running at level `function` of the call stack: Level 0 is the current function (`getinfo` itself); level 1 is the function that called `getinfo`; and so on. If `function` is a number larger than the number of active functions, then `getinfo` returns **nil**.

The returned table contains all the fields returned by `lua_getinfo`, with the string `what` describing which fields to fill in. The default for `what` is to get all information available. If present, the option `` `f´ `` adds a field named `func` with the function itself.

For instance, the expression `debug.getinfo(1,"n").name` returns the name of the current function, if a reasonable name can be found, and `debug.getinfo(print)` returns a table with all available information about the `print` function.

debug.getlocal (level, local)

This function returns the name and the value of the local variable with index `local` of the function at level `level` of the stack. (The first parameter or local variable has index 1, and so on, until the last active local variable.) The function returns **nil** if there is no local variable with the given index, and raises an error when called with a `level` out of range. (You can call `debug.getinfo` to check whether the level is valid.)

debug.getupvalue (func, up)

This function returns the name and the value of the upvalue with index `up` of the function `func`. The function returns **nil** if there is no upvalue with the given index.

debug.setlocal (level, local, value)

This function assigns the value `value` to the local variable with index `local` of the function at level `level` of the stack. The function returns **nil** if there is no local variable with the given index, and raises an error when called with a `level` out of range. (You can call `getinfo` to check whether the level is valid.)

debug.setupvalue (func, up, value)

This function assigns the value `value` to the upvalue with index `up` of the function `func`. The function returns **nil** if there is no upvalue with the given index.

debug.sethook (hook, mask [, count])

Sets the given function as a hook. The string `mask` and the number `count` describe when the hook will be called. The string mask may have the following characters, with the given meaning:

- **"c"** The hook is called every time Lua calls a function;
- **"r"** The hook is called every time Lua returns from a function;
- **"l"** The hook is called every time Lua enters a new line of code.

With a `count` different from zero, the hook is called after every `count` instructions.

When called without arguments, the `debug.sethook` function turns off the hook.

When the hook is called, its first parameter is always a string describing the event that triggered its call: `"call"`, `"return"` (or `"tail return"`), `"line"`, and `"count"`. Moreover, for line events, it also gets as its second parameter the new line number. Inside a hook, you can call `getinfo` with level 2 to get more information about the running function (level 0 is the `getinfo` function, and level 1 is the hook function), unless the event is `"tail return"`. In this case, Lua is only simulating the return, and a call to `getinfo` will return invalid data.

debug.traceback ([message])

Returns a string with a traceback of the call stack. An optional `message` string is appended at the beginning of the traceback. This function is typically used with `xpcall` to produce better error messages.

6 - Lua Stand-alone

Although Lua has been designed as an extension language, to be embedded in a host C program, it is also frequently used as a stand-alone language. An interpreter for Lua as a stand-alone language, called simply `lua`, is provided with the standard distribution. The stand-alone interpreter includes all standard libraries plus the reflexive debug interface. Its usage is:

```
lua [options] [script [args]]
```

The options are:

- `-` executes `stdin` as a file;
- `-e` *stat* executes string *stat*;
- `-l` *file* "requires" *file*;
- `-i` enters interactive mode after running *script*;
- `-v` prints version information;
- `--` stop handling options.

After handling its options, `lua` runs the given *script*, passing to it the given *args*. When called without arguments, `lua` behaves as `lua -v -i` when `stdin` is a terminal, and as `lua -` otherwise.

Before running any argument, the interpreter checks for an environment variable `LUA_INIT`. If its format is *@filename*, then lua executes the file. Otherwise, lua executes the string itself.

All options are handled in order, except `-i`. For instance, an invocation like

```
$ lua -e'a=1' -e 'print(a)' script.lua
```

will first set `a` to 1, then print `a`, and finally run the file `script.lua`. (Here, `$` is the shell prompt. Your prompt may be different.)

Before starting to run the script, `lua` collects all arguments in the

command line in a global table called `arg`. The script name is stored in index 0, the first argument after the script name goes to index 1, and so on. The field `n` gets the number of arguments after the script name. Any arguments before the script name (that is, the interpreter name plus the options) go to negative indices. For instance, in the call

```
$ lua -la.lua b.lua t1 t2
```

the interpreter first runs the file `a.lua`, then creates a table

```
arg = { [-2] = "lua", [-1] = "-la.lua", [0] = "b.lua",
        [1] = "t1", [2] = "t2"; n = 2 }
```

and finally runs the file `b.lua`.

In interactive mode, if you write an incomplete statement, the interpreter waits for its completion.

If the global variable `_PROMPT` is defined as a string, then its value is used as the prompt. Therefore, the prompt can be changed directly on the command line:

```
$ lua -e"_PROMPT='myprompt> '" -i
```

(the outer pair of quotes is for the shell, the inner is for Lua), or in any Lua programs by assigning to `_PROMPT`. Note the use of `-i` to enter interactive mode; otherwise, the program would end just after the assignment to `_PROMPT`.

In Unix systems, Lua scripts can be made into executable programs by using `chmod +x` and the `#!` form, as in

```
#!/usr/local/bin/lua
```

(Of course, the location of the Lua interpreter may be different in your machine. If `lua` is in your `PATH`, then

```
#!/usr/bin/env lua
```

is a more portable solution.)

Acknowledgments

Incompatibilities with Previous Versions

Lua 5.0 is a major release. There are several incompatibilities with its previous version, Lua 4.0.

Incompatibilities with version 4.0

Changes in the Language

- The whole tag-method scheme was replaced by metatables.

- Function calls written between parentheses result in exactly one value.

- A function call as the last expression in a list constructor (like `{a,b,f()}`) has all its return values inserted in the list.

- The precedence of **or** is smaller than the precedence of **and**.

- **in**, **false**, and **true** are reserved words.

- The old construction `for k,v in t`, where `t` is a table, is deprecated (although it is still supported). Use `for k,v in pairs(t)` instead.

- When a literal string of the form `[[...]]` starts with a newline, this newline is ignored.

- Upvalues in the form `%var` are obsolete; use external local variables instead.

Changes in the Libraries

- Most library functions now are defined inside tables. There is a compatibility script (`compat.lua`) that redefines most of them as global names.

- In the math library, angles are expressed in radians. With the compatibility script (`compat.lua`), functions still work in degrees.

- The `call` function is deprecated. Use `f(unpack(tab))` instead of `call(f, tab)` for unprotected calls, or the new `pcall` function for protected calls.

- `dofile` does not handle errors, but simply propagates them.

- `dostring` is deprecated. Use `loadstring` instead.

- The `read` option `*w` is obsolete.

- The `format` option `%n$` is obsolete.

Changes in the API

- `lua_open` does not have a stack size as its argument (stacks are dynamic).

- `lua_pushuserdata` is deprecated. Use `lua_newuserdata` or `lua_pushlightuserdata` instead.

The Complete Syntax of Lua

chunk ::= {stat [`;´]}

block ::= chunk

stat ::= varlist1 `=´ explist1 | functioncall | **do** block **en**

funcname ::= Name {`.´ Name} [`:´ Name]

varlist1 ::= var {`,´ var}

var ::= Name | prefixexp `**[**´ exp `**]**´ | prefixexp `.´ Name

namelist ::= Name {`,´ Name}

init ::= `**=**´ explist1

explist1 ::= {exp `,´} exp

exp ::= **nil** | **false** | **true** | Number | Literal | function |

prefixexp ::= var | functioncall | `**(**´ exp `**)**´

functioncall ::= prefixexp args | prefixexp `:´ Name args

args ::= `**(**´ [explist1] `**)**´ | tableconstructor | Literal

function ::= **function** funcbody

```
funcbody ::= `(´ [parlist1] `)´ block end


parlist1 ::=  Name {`,´ Name} [`,´ `...´] | `...´


tableconstructor ::= `{´ [fieldlist] `}´
fieldlist ::= field {fieldsep field} [fieldsep]
field ::= `[´ exp `]´ `=´ exp | name `=´ exp | exp
fieldsep ::= `,´ | `;´


binop ::= `+´ | `-´ | `*´ | `/´ | `^´ | `..´ | `<´ | `<=´ |


unop ::= `-´ | not
```

Topic Group (Programming with LuaEdit)

- [Integrated Development Environment (IDE)](#)
- [Working with Projects](#)
- [Basics](#)
- [Editor Settings](#)
- [Editing Code](#)
- [Debugging Scripts](#)