International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

H.766

(08/2018)

SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS

IPTV multimedia services and applications for IPTV – IPTV multimedia application frameworks

Lua for IPTV services

Recommendation ITU-T H.766

International Telecommunication Union

ITU-T H-SERIES RECOMMENDATIONS

AUDIOVISUAL AND MULTIMEDIA SYSTEMS

CHARACTERISTICS OF VISUAL TELEPHONE SYSTEMS	H.100–H.199
INFRASTRUCTURE OF AUDIOVISUAL SERVICES	
General	H.200–H.219
Transmission multiplexing and synchronization	H.220–H.229
Systems aspects	H.230–H.239
Communication procedures	H.240–H.259
Coding of moving video	H.260–H.279
Related systems aspects	H.280–H.299
Systems and terminal equipment for audiovisual services	H.300–H.349
Directory services architecture for audiovisual and multimedia services	H.350–H.359
Quality of service architecture for audiovisual and multimedia services	H.360–H.369
Telepresence, immersive environments, virtual and extended reality	H.420–H.439
Supplementary services for multimedia	H.450–H.499
MOBILITY AND COLLABORATION PROCEDURES	
Overview of Mobility and Collaboration, definitions, protocols and procedures	H.500–H.509
Mobility for H-Series multimedia systems and services	H.510–H.519
Mobile multimedia collaboration applications and services	H.520–H.529
Security for mobile multimedia systems and services	H.530–H.539
Security for mobile multimedia collaboration applications and services	H.540–H.549
VEHICULAR GATEWAYS AND INTELLIGENT TRANSPORTATION SYSTEMS (ITS)	
Architecture for vehicular gateways	H.550–H.559
Vehicular gateway interfaces	H.560–H.569
BROADBAND, TRIPLE-PLAY AND ADVANCED MULTIMEDIA SERVICES	
Broadband multimedia services over VDSL	H.610–H.619
Advanced multimedia services and applications	H.620–H.629
Ubiquitous sensor network applications and Internet of Things	H.640–H.649
IPTV MULTIMEDIA SERVICES AND APPLICATIONS FOR IPTV	
General aspects	H.700–H.719
IPTV terminal devices	H.720–H.729
IPTV middleware	H.730–H.739
IPTV application event handling	H.740–H.749
IPTV metadata	H.750–H.759
IPTV multimedia application frameworks	**H.760–H.769**
IPTV service discovery up to consumption	H.770–H.779
Digital Signage	H.780–H.789
E-HEALTH MULTIMEDIA SYSTEMS, SERVICES AND APPLICATIONS	
Personal health systems	H.810–H.819
Interoperability compliance testing of personal health systems (HRN, PAN, LAN, TAN and WAN)	H.820–H.859
Multimedia e-health data exchange services	H.860–H.869
Safe listening	H.870–H.879

For further details, please refer to the list of ITU-T Recommendations.

Recommendation ITU-T H.766

Lua for IPTV services

Summary

Recommendation ITU-T H.766 specifies Lua as one of the standard multimedia application frameworks for Internet protocol television (IPTV) services. Lua is an extension programming language designed to support general procedural programming with data description facilities. Lua is the scripting language for the Ginga-nested context language (Ginga-NCL) in Recommendation ITU-T H.761. The Lua engine is distributed as free software under the MIT licence.

Lua for IPTV services is structured in two sets of application programming interfaces (APIs). The Lua IPTV core API is a basic, mandatory API that conforms with Ginga-NCL in Recommendation ITU-T H.761. The Lua IPTV extended API includes enhanced functionality that a Lua implementation is recommended to support. Recommendation ITU-T H.766 specifies the Lua IPTV core API and the Lua IPTV extended API. Recommendation ITU-T H.766 includes an annex with a reference manual for Lua 5.1 as well as an appendix with sample Lua code that implements a simple health application.

History

Edition	Recommendation	Approval	Study Group	Unique ID*
1.0	ITU-T H.766	2018-08-29	16	11.1002/1000/13671

* To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, http://handle.itu.int/11.1002/1000/11830-en.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at http://www.itu.int/ITU-T/ipr/.

Table of Contents

Page

Page

Page

Recommendation ITU-T H.766

Lua for IPTV services

1 Scope

This Recommendation specifies the Lua scripting language to provide interoperability and harmonization among Internet protocol television (IPTV) multimedia application frameworks. To provide global standard IPTV services, it is foreseeable that a combination of different standard multimedia application frameworks will be used. Therefore, this Recommendation specifies the Lua language as one of those standard multimedia application frameworks to provide interoperable use of IPTV services.

Lua for IPTV services is structured in two application programming interfaces (APIs). The Lua IPTV core API is a basic, mandatory API that conforms to the Ginga-nested context language (Ginga-NCL) [ITU-T H.761], which is a presentation engine that integrates NCL and Lua players into a declarative environment. The Lua IPTV extended API includes enhanced functionality that a Lua implementation is recommended to support.

2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this document. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this document are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

[ITU-T H.761] Recommendation ITU-T H.761 (2014), *Nested context language (NCL) and Ginga-NCL*.

[ITU-T H.770] Recommendation ITU-T H.770 (2015), *Mechanisms for service discovery and selection for IPTV services*.

3 Definitions

3.1 Terms defined elsewhere

This Recommendation uses the following terms defined elsewhere:

3.1.1 application programming interface (API) [b-ITU-T J.200]: Software libraries that provide uniform access to system services.

3.1.2 scripting language [ITU-T H.761]: The language used to describe an imperative object content that is embedded in another host language. For example, Lua is a scripting language for NCL documents as ECMAScript is a scripting language for HTML documents.

3.2 Terms defined in this Recommendation

This Recommendation defines the following terms:

3.2.1 Lua event class: A group of events that can be generated or handled by Lua scripts. Each group is usually related to a specific functionality of an Internet protocol television (IPTV) terminal device.

3.2.2 Lua Internet protocol television core application programming interface: The set of Lua functions and event classes that a Lua implementation is required to support for basic IPTV functionality.

3.2.3 Lua Internet protocol television extended application programming interface: The set of Lua functions and event classes that a Lua implementation is recommended to support for extended IPTV functionality.

3.2.4 Lua module: A group of functions that can be used by Lua scripts. Each group is usually related to a specific functionality of an IPTV terminal device.

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations and acronyms:

API Application Programming Interface

ASCII American Standard Code for Information Interchange

BNF Backus-Naur Form

HTTP Hypertext Transfer Protocol

IPTV Internet Protocol Television

NCL Nested Context Language

PVR Personal Video Recorder

RTP Real-time Transport Protocol

SDS Service Discovery and Selection

SMS Short Message Service

TCP Transmission Control Protocol

UDP User Datagram Protocol

5 The Lua scripting language

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 lightweight scripting language for any program that needs one. Lua is implemented as a library, written in clean C (i.e., 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 may invoke functions to execute a piece of Lua code, may write and read Lua variables, and may register C functions to be called by Lua code. Through the use of C functions, Lua may 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 sample host program called lua, which uses the Lua library to offer a complete, stand-alone Lua interpreter.

Annex A presents the Lua specification as a general-purpose language. However, any conformant implementation of Lua for IPTV services shall follow the restrictions specified in clause 6 and shall provide the Lua IPTV core API described in clause 8.

The Lua IPTV extended API described in clause 9, in its turn, is optional, but shall be followed if any of its functionalities is to be implemented.

6 Lua restricted functionalities

6.1 Functions removed from Lua library

The following functions are platform dependent and shall be removed from Lua libraries (see clause A.4) in conformant implementations:

1) from module *package*: *loadlib*;
2) from module *os*: *clock, execute, exit, getenv, remove, rename, tmpname* and *setlocale*;
3) from module *debug*: all functions.

7 Execution model

Lua, as an extension programming language, is intended to be used as a powerful, lightweight scripting language for any program that needs one. Lua scripts are commonly used as a piece of code embedded into another program, written in another language. Therefore, Lua scripts may have different execution models, depending on the host program interpreter. Refer to the specification of the host program interpreter (e.g. Ginga-NCL [ITU-T H.761]) for information about the Lua script lifecycle and available controlling interfaces.

8 Lua IPTV core API

The Lua IPTV core API includes all basic functionality a conformant implementation shall provide. Besides the Lua standard library (restricted as described in clause 6) the following Lua modules shall be present and automatically loaded in a conformant implementation:

1) canvas: offers an API to draw basic graphical components and manipulate images;
2) event: allows Lua applications to communicate with the middleware through events;
3) persistent: exports a table with persistent variables that may be manipulated only by imperative objects.
4) security module: responsible for offering functionalities related to data security.

The following event classes shall be supported in a conformant implementation:

1) key class: allows for handling of events generated by input devices;
2) pointer class: allows for handling of events generated by pointer devices, e.g., a mouse;
3) tcp, udp and http classes: allows for event-driven communication using the transmission control protocol (TCP), user datagram protocol (UDP) and hypertext transfer protocol (HTTP), respectively;
4) metadata class: allows for event-driven access to metadata information;
5) user class: allows applications to extend their functionality by creating their own events.

8.1 The canvas module

A canvas offers a graphical API to be used in a Lua application. Using this API, it is possible to draw lines, rectangles, text, images, etc.

A canvas keeps in its state a set of attributes under which the drawing primitives operate. For instance, if its colour attribute is blue, a call to `canvas:drawLine()` will draw a blue line on the canvas.

A global variable called *canvas* is available after the initialization of a Lua application, representing a graphic region with size and coordinates equal to the display information given to the Lua interpreter.

The canvas module is mandatory for the Lua IPTV core API and must be in conformance with the canvas module as specified in clause 10.3.1 of [ITU-T H.761].

8.2 The event module

This module offers an API for event handling. Using the API, a Lua application may communicate with many middleware components asynchronously.

An application may also use this mechanism internally, using the "user" event class.

The typical use of Lua application is to handle events: Middleware-generated events or events coming from user interactions (e.g., through the remote control).

During its initiation, before becoming event oriented, a Lua script must register an event handler function. After the initialization, any action performed by the script will be in response to an event notified to the application, i.e., to the event handler function.

The event module is mandatory for the Lua IPTV core API and must be in conformance with the event module as specified in clause 10.3.2 of [ITU-T H.761].

8.3 The persistent module

Lua applications may save data in a restricted middleware area and recover it between executions. The Lua player shall allow an application to persist a value to be used by itself or by another imperative object. In order to do that it defines a reserved area, inaccessible to non-imperative objects. This area is split into the groups "service", "channel" and "shared". There are no predefined or reserved variables in these groups, and imperative objects are allowed to change variable values directly. Other imperative languages should offer an API to access this same area.

In this module, Lua offers an API to export the *persistent* table with the variables defined in the reserved area.

The persistent module is mandatory in the Lua IPTV core API and must be in conformance with the persistent module as specified in clause 10.3.4 of [ITU-T H.761].

8.4 The security module

The objects that are part of the security module are responsible for offering functionalities related to data security, e.g., digital signature generation and verification, message digest generation, as well as mechanisms for data encryption.

The security module is mandatory in the Lua IPTV core API and must be in conformance with the security module as specified in clause 11.2 of [ITU-T H.761].

8.5 Lua event classes

8.5.1 The key class

The key class allows for handling of events generated by input devices.

The key class is an event class mandatory in the Lua IPTV core API and must be in conformance with the key class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.2 The pointer class

The pointer class allows for handling of events generated by pointer devices, such as mouse.

The pointer class is an event class mandatory in the Lua IPTV core API and must be in conformance with the pointer class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.3 The udp class

The udp class allows event-driven communication using the UDP.

The udp class is an event class mandatory in the Lua IPTV core API and must be in conformance with the udp class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.4 The tcp class

The tcp class allows event-driven communication using the TCP.

The tcp class is an event class mandatory in the Lua IPTV core API and must be in conformance with the tcp class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.5 The http class

The http class allows event-driven communication using the HTTP.

The http class is an event class mandatory in the Lua IPTV core API and must be in conformance with the http class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.6 The metadata class

The metadata class allows event-driven access to metadata information.

The metadata class is an event class mandatory in the Lua IPTV core API and must be in conformance with the metadata class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.7 The user class

The user class allows applications to extend their functionality by creating their own events.

The user class is an event class mandatory in the Lua IPTV core API and must be in conformance with the user class as specified in clause 10.3.2.2 of [ITU-T H.761].

9 Lua IPTV extended API

The Lua IPTV extended API includes enhanced functionality a conformant implementation may provide. If implemented, the functionality described in this clause must be in conformance with the following API specification.

The Lua IPTV extended API includes the following Lua modules:

1) pvr module: offers an API to control the personal video recorder (PVR) functionality;
2) widgets module: offers an API to draw graphical widgets like buttons and input text boxes.

The following event classes shall be supported in a conformant implementation:

1) extended tcp class: allows for extended, event-driven communication using the TCP;
2) extended udp class: allows for extended, event-driven communication using UDP protocol;
3) rtp class: allows for event-driven communication using the real-time transport protocol (RTP);
4) sms class: allows for event-driven communication using the short message service (SMS) protocol;
5) sds class: allows applications to receive events from the service discovery and selection (SDS) [ITU-T H.770] functionality;
6) smartcard class: allows for handling of events generated by smartcard devices;
7) pvr class: allows for handling of events generated by PVR operations;
8) widget class: allows for handling of events generated by graphical widgets.

Lua modules and event classes mentioned in the foregoing are detailed in the following clauses. The definition of each function uses the following naming convention:

```
funcname (parnameI: partypeI [; optnameI: opttypeI]) -> retname: rettype
```

9.1 The pvr module

The pvr module offers support for PVR control for IPTV systems.

9.1.1 Constructors

pvr:new (filename: string) -> pvr: object

Arguments

Filename: Filename path (audio/video stream)

Return values

pvr New pvr object

Description

Returns a new pvr object, which is used to control the PVR functionality. If an error occurs, the pvr object is returned with the attribute attrError different from 0 and all attempts to call methods of this object will return the error PVR_ERROR_CREATE.

9.1.2 Attributes

pvr:attrError() -> errorCode: number

Arguments

none

Return values

errorCode – The error code. The possible values are:

```
-1  - PVR_ERROR_CREATE
-2  - PVR_ERROR_DEVICE_NOT_PRESENT
-3  - PVR_ERROR_PERMISSION_DENIED
-4  - PVR_ERROR_NOT_ENOUGH_SPACE
-5  - PRV_ERROR_UNKOWN_SERVICEID
-6  - PVR_ERROR_INVALID_DATE_TIME
-7  - PVR_ERROR_INVALID_OPERATION_ID
-8  - PVR_ERROR_CANCELLING_OPERATION
-10 - PVR_ERROR_UNKOWN
```

Description

Returns an error code representing an error in the last PVR operation

9.1.3 Primitives

pvr:schedule ([datetime:table]; [deviation], duration, serviceId: number) -> operationId: number

Arguments

datetime: the recording start time, as an absolute date/time. It is a table value with the following fields: year, month (1-12), day (1-31), hour (0-23), min (0-59), sec (0-59), and isdst (a daylight saving flag, boolean). If omitted, it means immediately.

deviation: number of seconds to anticipate (negative number) or defer (positive number) the recording start time. If omitted or equals to 0 (zero), it means no time deviation.

duration: duration of recording operation in seconds

serviceId: service Id that shall be recorded

Return values

operationId representing the pvr record. It may also indicate an error if the operation failed. The possible errors are:

```
PVR_ERROR_INVALID_DATE_TIME
PVR_ERROR_UNKNOWN_SERVICEID
PVR_ERROR_PERMISSION_DENIED
PVR_ERROR_UNKOWN
```

Description

Schedules a PVR recording operation. If datetime and deviation are omitted, then recording starts immediately. This can be also used for time shifting operation where the playback media becomes the recorded file instead of the received transport stream.

pvr:getList() -> pvrList: table

Return values

Table containing all active pvr operations

```
pvrList = {
      {
         pvrOperationId   = <number>,
         startDateTime    = <table>,
         duration         = <number>,
         serviceId        = <number>,
      },...
          }
```

Description

Lists all scheduled PVR operations. startDateTime represents the effective recording start time, considered a specified deviation (see pvr:schedule()).

pvr:cancel (pvrOperationId: number) -> errorCode: number

Arguments

pvrOperationId – id of the pvr operation to be cancelled

Return values

errorCode representing an error in the cancel operation. The possible values are:

```
PVR_ERROR_INVALID_OPERATION_ID
PVR_ERROR_CANCELLING_OPERATION
PVR_ERROR_PERMISSION_DENIED
PVR_ERROR_UNKNOWN
```

Description

Cancel a scheduled PVR operation

9.2 The widgets module

The widgets module offers drawing primitives to facilitate the design of richer graphical components, such as buttons and input text boxes, which are not supported by the Lua canvas module. This module also offers cursor handling support and mechanisms for creating and scrolling a virtual canvas, which is a canvas with a larger area than that defined for a Lua application canvas.

9.2.1 Object widget

The widget object is responsible for filling an optional given canvas with a graphic element (widget) and also for posting input events to the registered handlers.

9.2.1.1 Constructors

widget:new (canvas: object)-> widget: object

Arguments:

canvas – a canvas object to be associated with this widget object. This canvas can have an area larger than that defined for its Lua application, therefore it is possible to scroll over it.

Return values:

widget – a new widget object

Description

Returns a new widget object.

9.2.1.2 Attributes

widget:attrCanvas(canvas: object)

Arguments:

canvas – Sets the canvas object used to display the graphic element of this widget.

Description

Sets the canvas object related to this widget.

widget:attrCanvas()->canvas: object

Return Values:

canvas – a canvas object that will be related to this widget.

Description

Retrieves the associated canvas object of this widget.

9.2.1.3 Miscellaneous

widget:scroll(x,y: number)

Arguments:

x,y – number values used for scrolling the associated canvas.

Description

Scrolls the widget canvas to a given position (identified by two integer values, x and y) by composing it along with the application canvas.

widget:addTooltip(x,y,w,h: number, msg: string)-> tooltipID: number

Arguments:

x,y,w,h – number values defining the area of activation for this tooltip.

msg – a string containing the tooltip message.

Return Values:

tooltipID – A unique number value for this tooltip identification

Description

Adds a tooltip to the widget, to be displayed when the cursor stops over the specified area.

widget:removeTooltip(tooltipID: number)

Arguments:

tooltipID – a unique number value identifier for a tooltip, generated by addTooltip function.

Description

Remove the tooltip identified by the value of tooltipID.

9.2.2 Object textinput

The textinput object is responsible for offering a mechanism for capturing text from the user input, so that it can be eventually dispatched to a widget object. The textinput object extends the event module input handling by adding support for modifiers keys (e.g. CTRL+C, SHIFT+A).

9.2.2.1 Constructors

textinput:new ( )-> textinput: object

Return values:

textinput – a new textinput object

Description

Returns a new textinput object, which is a graphical component that allows users to input text. This object will dispatch the events generated by user input interaction to the registered handlers.

textinput:new (canvas: object)-> textinput: object

Arguments:

canvas – the canvas where the widget graphical component will be rendered

Return values:

textinput – a new textinput object

Description

Returns a new textinput object that allows a Lua application to handle text input. This object will dispatch the events generated by user input interaction to registered handlers. The canvas object parameter is used to display the captured text.

The textinput will be formatted according to the attributes of its canvas, e.g., font (attrFont) or color (attrColor).

9.2.2.2 Attributes

textinput:attrCanvas(canvas: object)

Arguments:

canvas – Sets the canvas object to be associated with this textinput, so that it is able to display the entered text on this canvas.

Description

Sets the canvas object to be associated with this textinput, in order to display the entered text on the canvas.

textinput:attrCanvas()->canvas: object *Return Values:* canvas – a canvas object to be associated with this textinput. *Description* Retrieves the associated canvas object of this textinput.
textinput:attrText(text: string) *Arguments:* text – a string containing the text to fill the textinput. *Description* Sets a text that fills the textinput space, which can later be changed by the user.
textinput:attrText()->text: string *Return Values:* text – a string containing the textinput current text. *Description* Returns the current text filled in this object.
textinput:attrEnable(enable: boolean) *Arguments:* enable – a boolean value to define whether this widget is active. *Description* Sets the status of this widget to active (able to be filled by the user) or inactive (blocked for user input).
textinput:attrEnable()->enable: Boolean *Return Values:* enable – a boolean with the current activation status of this widget. *Description* Returns the current activation status of this widget.

9.2.3 Object cursor

A cursor object is a pointer cursor (associated with an image that is used as its graphical representation), which is exhibited over a canvas.

When the supporting platform is provided with a pointing device, one instance of this object is automatically created for a Lua application. Lua applications can independently customize this object (e.g., change the cursor image) while inside its exhibiting canvas.

9.2.3.1 Attributes

cursor:attrImage (path: string) *Arguments:* path – a string containing the image source path. *Description* Sets the source URL of a cursor image to a given string value, replacing the cursor image.
cursor:attrImage()->path: string *Return Values:* path – the source path string for the image being used as a pointer cursor. *Description* Returns the current source path for the cursor image.
cursor:attrPosition(x,y: number) *Arguments:* x,y – two numbers to define the new positioning of a cursor in the canvas. *Description* Sets the position of the cursor to a given coordinate (expressed by the x and y numbers) in the canvas. A `WIDGET_CURSOROVER` event is launched after cursor positioning.
cursor:attrPosition()->x,y: number *Return Values:* x,y – two numbers containing the current coordinates of a cursor in the canvas. *Description* Returns the current coordinates of the cursor in the canvas.
cursor:attrEnable(enable: boolean) *Arguments:* enable – a boolean defining whether the cursor is active. *Description* Sets the status of the cursor to active or inactive. An active cursor is rendered on the screen according to its attributes and allows for user. An inactive cursor is hidden.
cursor:attrEnable()->enable: Boolean *Return Values:* enable – a boolean with the current status of the cursor. *Description* Returns the current status of the cursor (active or inactive)

9.2.3.2 Miscellaneous

cursor:click(button: number)

Arguments:

button – a number value identifying the button to be clicked. See the possible values in clause 9.3.8.

Description

Performs a single click on the current position the cursor is over and generates two events of the class 'widget', type cursor, subtypes 'cursor_press' and 'cursor_release'. See clause 9.3.8.

cursor:press(button: number)

Arguments:

button – a number value identifying the button to be pressed. See the possible values in clause 9.3.8.

Description

Presses the specified button on the current position the cursor is over and generates an event of the class 'widget', type cursor, subtype 'cursor_press'. See clause 9.3.8.

cursor:release(button: number)

Arguments:

button – a number value identifying the button to be released. See the possible values in clause 9.3.8.

Description

Releases the specified button and generates an event of the class 'widget', type cursor, subtype 'cursor_release'. See clause 9.3.8.

9.3 Lua event classes

9.3.1 Extended tcp class:

The Lua extended tcp class allows for event-driven communication using the TCP

In order to send or receive TCP data in client-mode, a connection shall be firstly established through posting an event in the form:

```
evt = { class='tcp', type='connect', host=string, port=number,
          [interface_addr=string], [local_port=number],
          [timeout=number] }
```

where:
host is the IP address or name of the remote host
port is the TCP port of the remote host
interface_addr is the IP address of the local interface to be used. If not specified, the local interface to be used is assigned by the operating system
local_port is the local TCP port to be used. If not specified, the local port to be used is assigned by the operating system

TCP communications may also be initiated remotely and be awaited (server mode), which is started by posting an event in the form:

```
evt = { class='tcp', type='listen', [interface_addr=string],
          [local_port=number] }
```

In both cases, (type='connect' and type='listen'), the connection result is returned in a pre-registered event handler for the tcp class. When in server mode, the returned event is received each time a remote host establishes a new connection. The returned event has the following form:

```
evt = { class='tcp', type='connect', host=string, port=number,
          interface_addr=string, local_port=number,
          connection=number, error=string }
```

The *error* and *connection* fields are mutually exclusive. If there is a communication error, an error message is returned in the *error* field. If, however, the connection is successfully established, the connection identifier is returned in the *connection* field.

NOTE – If there is an attempt to establish a connection with a pair *interface_addr/local_port* already in use, the connection is unsuccessful and the *error* field must have the value 'port unavailable'.

A Lua application sends data, using the TCP, by posting events of the form:

```
evt = { class='tcp', type='data', connection=number,
          value=string, [timeout=number] }
```

Similarly, a Lua application receives data transported by the TCP by handling events of the form:

```
evt = { class='tcp', type='data', connection=number,
          value=string, error=string }
```

The *error* and *value* fields are mutually exclusive. If there is a communication error, an error message is returned in the *error* field. Otherwise, the message is passed in the *value* field.

In order to close the connection, an event of the following form shall be posted:

```
evt = { class='tcp', type='disconnect', connection=number }
```

Similarly, if the remote host closes the connection or the connection is lost, an event of the following form is returned:

```
evt = { class='tcp', type='disconnect', connection=number, error=string }
```

In order to finish the acceptance for new TCP connections (server mode), a Lua application must post an event of the form:

```
evt = { class='tcp', type='unlisten', local_port=number }
```

The class dependent filter of the "tcp" class is *connection*.

9.3.2 Extended udp class:

Lua extended udp class allows for event-driven communication using UDP protocol

A Lua application sends data, using UDP protocol, by posting events of the form:

```
evt = { class='udp', type='data', (host =string | broadcast=boolean),
          port=number, [interface_addr=string], [local_port=number],
          value=string }
```

where:

host is the IP address or name of the remote host

broadcast indicates that data must be sent to all hosts in the sub-network

port is the UDP port of the remote host

interface_addr is the IP address of the local interface to be used. If not specified, the local interface to be used is assigned by the operating system

local_port is the local TCP port to be used. If not specified, the local port to be used is assigned by the operating system

In order to receive UDP data, an association (bind) shall be firstly established through posting an event in the form:

```
evt = { class='udp', type='bind', [interface_addr=string],
          local_port=number, [max_data_length=number] }
```

where:

max_data_length specifies the maximum length in bytes for the value field in udp data reception events for the requested association.

An association result is returned in a pre-registered event handler for the class. The returned event has the form:

```
evt = { class='udp', type='bind', [interface_addr=string],
          local_port=number, association=number, error=string}
```

The *interface_addr* field is included in the returned event if the same field was specified in the bind event posted to establish the association.

The *error* and *association* fields are mutually exclusive. If there is an error, a message is returned in the *error* field. If, however, the association is successfully established, the association identifier is returned in the *association* field.

NOTE – If there is an attempt to establish an association with a pair *interface_addr/local_port* already in use, the association is unsuccessful and the *error* field must have the value 'port unavailable'.

A Lua application receives data transported by UDP protocol by handling events of the form:

```
evt = { class='udp', type='data', host=string, port=number,
          association=number, value=string }
```

In order to release an association, an event of the following form shall be posted:

```
evt = { class='udp', type='unbind', association=number }
```

The class dependent filter of the "udp" class is *association*.

9.3.3 The rtp class

The rtp class allows event-driven communication using the RTP.

A Lua script sends data, using the RTP, through posting events in the form:

```
evt = { class='rtp', host_addr=string, port=number,
          value=string, timeout=number }
```

Similarly, a Lua application receives data transported by RTP handling events in the form:

```
evt = { class='rtp', host_addr=string, port=number,
           value=string, error_msg=string}
```

The *error_msg* and *value* fields are mutually exclusive. When there is a communication error, a message is returned in the *error_msg* field. When the communication is succeeded, the message is passed in the *value* field.

NOTE 1 – A specific middleware implementation should handle issues like authentication, connection timeout/retry or whether a connection should remain open.

NOTE 2 – In the rtp class, the class dependent filter could only be *host_addr, port*.

9.3.4 The sms class

The sms class allows event-driven communication using the SMS protocol.

The sms class is an event class must be in conformance with the sms class as specified in clause 10.3.2.2 of [ITU-T H.761].

9.3.5 The sds class

The sds class allows applications to receive events from the SDS [ITU-T H.770] functionality.

sds class

event class sds provides access to the list of available service providers and their respective services. For detailed information about SDS mechanisms, refer to [ITU-T H.770].

The information acquisition process shall be performed in two steps:

1) A request is made calling the asynchronous `event.post()` function;

2) An event, to be delivered to the registered-event handlers of a Lua script, whose data field contains a set of subfields and is represented by a Lua table. The set of subfields depends on requested information.

NOTE 1 – In the sds class, the class dependent filter could only be *type*.

NOTE 2 – Any sds event may be received without any previous request. This shall happen in case of changes in the list of available service providers or on the list of the available services.

Six event types are defined by the following tables:

type = 'providers'

The table of 'providers' event type is composed of a set of vectors, each one with information related to a service provider.

Each request for a table of 'providers' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type='providers'}),
```

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```
evt = {
    class = 'sds',
    type = 'providers',
    serviceproviders = {
         {  -- serviceproviders is indexed like a vector
            serviceProviderIdentifier = <string>, -- Unique URL, domain name
   etc
            serviceProviderName = <string>,
            serviceProviderDescription  = <string>,
            serviceProvideLogoUri = <string>,
            serviceOfferSummary = {
               {   -- serviceOfferSummary is indexed like a vector
                   OfferType = <number>, -- type of the service


                   PushAddress = <string>
                   pullUrl = <string>
                   webPortalUrl = <string>

               }, ...
            },
            recordVersion = <number>
         }, ...
      }
   }
```

serviceType can have the following values:

```
0 - Linear TV Discovery
1 - Package Discovery
2 - Content Guide Discovery
3 - Service from Other Service Provider Discovery
1000 – Other services Discovery
```

type = 'linearTVServices'

The table of the 'linearTVServices' event type is used to receive all information concerning the available linear TV services.

Each request for a table of 'linearTVServices' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type= 'linearTVServices'[,
providerIdentifier] }),
```

where:

i) the *providerIdentifier* field is obtained from the table *serviceproviders*. If the *providerIdentifier* is not specified, then the list of linear TV services of all available service providers shall be requested.

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```
evt = {
    class = 'sds',
    type = 'linearTVServices',
    lineartvservices = {
         {  -- lineartvservices is indexed like a vector
            ServiceProviderIdentifier = <string>, -- Unique URL, domain name,
   etc

            metadataServerUrl = <string>,  -- if none value shall be nil and
                                           -- TS shall have SI/PSI
            PortalUrl = <string>,          -- if none value shall be nil
            purchaseInformationUrl = <string>,        -- if none value shall
   be nil
            serviceIdentifier = <string>,  -- unique identifier for the
   service
                                           -- in the service provider domain
            OriginalNetworkId = <number>,
            transportStreamId= <number>,
            serviceId = <number>,
            maxBitRate = <number>,          -- Max bitrate in kbits/s
            serviceLocations = { -- serviceLocations is indexed like a vector
               ipMulticastAddress = <string>, -- if none value shall be nil
               ipMulticastPort = <string>,
               ipMulticastSource = <string>,
               tvFec = {                      -- if none value shall be nil
                   fecBaseLayerMulticastAdddress = <string>,
                   fecEnhancementLayerMulticastAddress = <string>,
                   maximumPacketNumberInBlocks = <number>,
                   maxFecBlockDuration = <number>,
                   maxSourceBlockLength = <number>, -- only for fecEnhancement
                   encodingSymbolSize = <number>    -- only for fecEnhancement
                },
               unicastUrl = <string>,         -- if none value shall be nil
            }
            audioCoding = <number>,
```

```
            videoCoding = <number>,
            streamingType = <number>,  -- 0 RTP; 1 UDP; 1000 Other
            multiplexMode = <number>,   -- 0 TS; 1 Time-stamped TS; 1000 Other

            recordVersion = <number>
         }, ...
      }
   }
```

type = 'servicePackages'

The table of the 'servicePackages' event type is used to receive information of how the services are packaged together.

Each request for a table of 'servicePackages' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type= 'servicePackages'[,
providerIdentifier] }),
```

where:

i) the *providerIdentifier* field is obtained from the table *serviceproviders*. If the *providerIdentifier* is not specified, then the list of linear TV services of all available service providers shall be requested.

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```
evt = {
    class = 'sds',
    type = 'servicePackages',
    servicePackages = {
         {  -- servicePackages is indexed like a vector
            ServiceProviderIdentifier = <string>, -- Unique URL, domain name
   etc
            PackageIdentifier = <string>,
            displayable = <number>,
            packageName = <string>,
            packageDescription = <string>,
            preferedContentGuideIdentifier = <string>,
            individualservices = {
               {                -- inidividualservices is indexed like a
   vector
                  serviceProviderIdentifier = <string>, --if nil, then service
   provider is the same from service package
                  serviceIdentifier = <string>
               }, ...
            },
            recordVersion = <number>
         }, ...
      }
   }
```

type = 'contentGuideServices'

The table of the 'contentGuideServices' event type is used to receive information about content guide services.

Each request for a table of 'contentGuideServices' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type= 'contentGuideServices'[,
providerIdentifier] }),
```

where:

i) the *providerIdentifier* field is obtained from the table *serviceproviders*. If the *providerIdentifier* is not specified, then the list of linear TV services of all available service providers shall be requested.

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```
evt = {
    class = 'sds',
    type = 'contentGuideServices',
    contentguideservices = {
         {  -- contentguideservices is indexed like a vector
            ServiceProviderIdentifier = <string>, -- Unique URL, domain name
   etc
            contentGuideIdentifier = <string>,
            contentGuideName = <string>,
            contentGuideProviderName = <string>,
            contentGuideDescription = <string>,
            contentGuideLocator = <string>,
            contentGuideType = <number>, -- 0 linearTV; 1 vod; 2 both;3 other
            contentGuideLogoUri = <string>,
            targetServiceProviderIdentifier = <string>,

            recordVersion = <number>
         }, ...
      }
   }
```

type = 'thirdPartyServices'

The table of the 'thirdPartyServices' event type is used to receive information about third party services.

Each request for a table of 'thirdPartyServices' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type= 'thirdPartyServices'[,
providerIdentifier] }),
```

where:

i) the *providerIdentifier* field is obtained from the table *serviceproviders*. If the *providerIdentifier* is not specified, then the list of linear TV services of all available service providers shall be requested.

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```
evt = {
    class = 'sds',
    type = 'thirdPartyServices',
    thirdpartyservices = {
         {  -- thirdpartyservices is indexed like a vector
            ServiceProviderIdentifier = <string>, -- Unique URL, domain name
   etc
```

```
            referencedServiceProviderIdentifier = <string>, -- Unique URL,
   domain name etc
            individualServices = { -- if nil, then the entire set of offerings
   from the service provider is referenced
               {                  -- inidividualservices is indexed like a
   vector
                   serviceIdentifier = <string>
               }, ...
            recordVersion = <number>
         }, ...
      }
```

type = 'otherServices'

The table of the 'otherServices' event type is used to receive information about third party services.

Each request for a table of 'otherServices' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type= 'otherServices'[,
providerIdentifier] }),
```

where:

i) the *providerIdentifier* field is obtained from the table *serviceproviders*. If the *providerIdentifier* is not specified, then the list of linear TV services of all available service providers shall be requested.

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```
evt = {
    class = 'sds',
    type = 'otherServices',
    otherservices = {
         {                              -- otherservices is indexed like a
   vector
            ServiceProviderIdentifier = <string>, -- Unique URL, domain name
   etc
            serviceIdentifier = <string>,
            serviceName = <string>,
            serviceDescription = <string>,
            serviceType = <number>, -- 0 email; 1 SMS; 2 ebook; 1000 other
            serviceLocator = <string>,
                     recordVersion    = <number>
         }, ...
      }
   }
```

9.3.6 The smartcard class

The smartcard class allows handling of events generated by smartcard devices.

smartcard Class

```
     evt = {class = "smartcard", slot:number, eventId:number }
```

slot is the number of the smartcard slot the event refers to

eventId is the type of event. It can be:

```
     0 - CARD_INSERTED. Event when a card has been inserted
     1 - CARD_MUTE. Event when a card is in mute
```

```
2 - CARD_NOT_PRESENT. Event when no card is present
3 - CARD_REMOVED. Event when a card is removed
4 - CARD_TERMINAL_REMOVED. Event when the card terminal is removed
```

9.3.7 The pvr class

The pvr class allows handling of events generated by PVR operations.

pvr Class

```
evt = { class = "pvr",  eventId: number }
```

eventId is the type of event. It can be:

```
-3  - PVR_ERROR_PERMISSION_DENIED
-4  - PVR_ERROR_NOT_ENOUGH_SPACE
-10 - PVR_ERROR_UNKONW
5   - PVR_EVENT_5MIN_START  - indicates a recording starts in 5 minutes
6   - PVR_EVENT_RECORD_STOP - indicates the end of a recording operation
```

9.3.8 The widget class

The widget class allows handling of events generated by graphical widgets.

widget class:

The `widget` event class provides access to event information related to graphical widgets.

NOTE – In the widget class, the class dependent filter could only be *type*.

For any given type, the event is returned as follows:

```
evt = {
    class     = 'widget',
    type      = <string>,
    eventId   = <number>,
    sourceobj = <object>,
    data      = {...}
}
```

The following event types are defined:

type='textinput'

The table of the 'textinput' event type contains information about an asynchronous event occurred on a textinput widget, referred by *sourceobj*. The data field depends on event subtype and can encapsulate an inner table with key modifiers, e.g. "modifiers={alt, ctrl}" specifying any modifier keys pressed by the user.

eventId identifies an event subtype, as follows:

`01 - WIDGET_FOCUSED` – data field is nil

`02 - WIDGET_UNFOCUSED` – data field is nil

`11 - WIDGET_KEYPRESSED` – data field contains key value and modifiers – 'key="9"' modifiers={META}

`12 - WIDGET_KEYRELEASED` – data field contains key value and modifiers – 'key="9"' modifiers={SHIFT}

Possible values for the "modifiers" table are: META, ALT, CTRL, SHIFT, FN

type='cursor'

The table of the 'cursor' event type contains information about an asynchronous event occurred from a cursor widget, referred by *sourceobj*. The data field depends on event subtype.

eventId identifies an event subtype, as follows:

`01 - WIDGET_CURSOROVER` – data field contains cursor coordinates – 'x=12,y=105'

`02 - WIDGET_CURSORIN` – data field contains cursor coordinates – 'x=150,y=213'

03 – WIDGET_CURSOROUT – data field contains cursor coordinates – 'x=150,y=213'

11 – WIDGET_CURSORPRESS – data field contains cursor coordinates and a button identifier – 'x=150,y=213,button=1'

12 – WIDGET_CURSORRELEASE – data field contains cursor coordinates and button identifier – 'x=150,y=213,button=1'

13 – WIDGET_CURSORDOUBLECLICK – data field contains cursor coordinates and button identifier – 'x=150,y=213,button=1'

Possible values for the button identifier are:

01 – WIDGET LEFTBUTTON

02 – WIDGET_MIDDLEBUTTON

03 – WIDGET_RIGHTBUTTON

Annex A

Lua specification

(This annex forms an integral part of this Recommendation.)

The content of this annex was extracted from [b-Lua 5.1], and it is reprinted here with approval of the authors.

A.1 The language

A.1.1 Notation used

The language constructs are explained using the usual extended Backus–Naur form (BNF) notation, in which {a} means 0 or more instances of a, and [a] means an optional a. Keywords are shown in bold, non-terminals are shown in the standard document font, and other terminal symbols are also shown in the standard document font, but enclosed in single quotes. The complete syntax of Lua appears in clause A.7.

A.1.2 Lexical conventions

Names (also called identifiers) in Lua may be any string of letters, digits and underscores not beginning with a digit. This coincides with the definition of names in most languages. (The definition of letter depends on the current locale: any character considered alphabetic in the current locale may be used in an identifier.) Identifiers are used to name variables and table fields.

The keywords in Table A.1 are reserved and shall not be used as names.

Table A.1

`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 names. As a convention, names starting with an underscore followed by uppercase letters (such as _VERSION) are reserved for internal global variables used by Lua.

The following strings denote other tokens:

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

Literal strings may be delimited by matching single or double quotes, and may 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 (double quote)

– \' --- apostrophe (single quote)

Moreover, a `\newline´ (i.e., 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 digits. (Note that if a numerical escape is to be followed by a digit, it must be expressed using exactly three digits.) Strings in Lua may contain any 8-bit value, including embedded zeros, which may be specified as `\0´.

To put a double (single) quote, a newline, a backslash or an embedded zero inside a literal string enclosed by double (single) quotes, an escape sequence is required. Any other character may be directly inserted into the literal. (Some control characters can cause problems for the file system, but Lua has no problem with them.)

Literal strings may also be defined using a long format enclosed by long brackets. An opening long bracket of level n is defined as an opening square bracket followed by n equal signs followed by another opening square bracket. So, an opening long bracket of level 0 is written as [[, an opening long bracket of level 1 is written as [=[, and so on. A closing long bracket is defined similarly; for instance, a closing long bracket of level 4 is written as]====]. A long string starts with an opening long bracket of any level and ends at the first closing long bracket of the same level. Literals in this bracketed form can run for several lines, do not interpret any escape sequences, and ignore long brackets of any other level. They may contain anything except a closing bracket of the proper level or embedded zeros.

For convenience, when the opening long bracket is immediately followed by a newline, the newline is not included in the string. As an example, in a system using American standard code for information interchange (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)     "alo\n123\""
(3)     '\97lo\10\04923"'
(4)     [[alo
        123"]]
(5)     [==[
        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 with a double hyphen (--) anywhere outside a string. If the text immediately after -- is not an opening long bracket, 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 closing long bracket. Long comments are frequently used to disable code temporarily.

A.1.3 Values and types

A.1.3.1 Basic 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.

All values in Lua are first-class values. That means that all values may be stored in variables, passed as arguments to other functions and returned as results.

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; it usually represents the absence of a useful value. Boolean is the type of the values: false or true. 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. See file luaconf.h [b-luaconf.h].) String represents arrays of characters. Lua is 8-bit clean: Strings may contain any 8-bit character, including embedded zeros (`\0´) (see clause A.1.2).

Lua may call (and manipulate) functions written in Lua and functions written in C (see clause A.1.6.9).

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 may define operations for userdata values (see clause A.1.9). Userdata values shall not 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 (see clause A.1.12). Do not confuse Lua threads with operating-system threads. Lua supports coroutines on all systems, even those that do not support threads.

The type table implements associative arrays, i.e., arrays that may be indexed not only with numbers, but also with any value (except nil). Tables may be heterogeneous; i.e., they may 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 clause A.1.6.8).

Like indices, the value of a table field may 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 clause A.1.6.10).

Strings, 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.

A.1.3.2 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 conversion 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 over how numbers are converted to strings, use the format function from the string library (see string.format).

A.1.4 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 may denote a global variable or a local variable (or a function formal parameter, which is a particular kind of local variable):

```
var ::= Name
```

Name denotes identifiers, as defined in clause A.1.2.

Variables are assumed to be global unless explicitly declared local (see clause A.1.5.8). Local variables are lexically scoped: local variables can be freely accessed by functions defined inside their scope (see clause A.1.7).

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 in that table. The expression denoting the table to be indexed has a restricted syntax (see clause A.1.6.1).

The syntax var.Name is just syntactic sugar for var["Name"] and is used to denote table fields:

```
var ::= prefixexp `.´ Name
```

The meaning of accesses to global variables and table fields may be changed via metatables. An access to an indexed variable t[i] is equivalent to a call gettable_event(t,i). (See clause A.1.9 for a complete description of the gettable_event function. This function is not defined or callable in Lua. It is used here only for explanatory purposes.)

All global variables live as fields in ordinary Lua tables, called environment tables or simply environments (see clause A.1.10). Each 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 get the environment table of a Lua function, getfenv is called. To replace it, setfenv is called. [The environment of C functions may only be manipulated through the debug library (see clause A.4.10).]

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. (See clause A.1.9 for a complete description of the gettable_event function. This function is not defined or callable in Lua. Similarly, the _env variable is not defined in Lua. They are used here only for explanatory purposes.)

A.1.5 Statements

A.1.5.1 Basic concepts

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

A.1.5.2 Chunks

The unit of execution of Lua is called a chunk. A chunk is simply a sequence of statements that is executed sequentially. Each statement may be optionally followed by a semicolon:

```
chunk ::= {stat [`;´]}
```

There are no empty statements and thus `;;´ is not legal.

Lua handles a chunk as the body of an anonymous function with a variable number of arguments (see clause A.1.6.10). As such, chunks may define local variables, receive arguments 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 instructions 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 [b-luac] for details. Programs in source and compiled forms are interchangeable; Lua automatically detects the file type and acts accordingly.

A.1.5.3 Blocks

A block is a list of statements; syntactically, a block is the same as 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 clause A.1.5.5).

A.1.5.4 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 A.1.6.

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 instances of nil as needed. If the list of expressions ends with a function call, then all values returned by that call enter the list of values, before the adjustment (except when the call is enclosed in parentheses; see clause A.1.6).

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 may be changed via metatables. An assignment to an indexed variable t[i] = val is equivalent to settable_event(t,i,val) (see clause A.1.9 for a complete description of the settable_event function. This function is not defined or callable in Lua. It is used 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. It is used here only for explanatory purposes.)

A.1.5.5 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 block] end
```

Lua also has a for statement, in two flavours (see clause A.1.5.6).

The condition expression 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).

In the repeat--until loop, the inner block does not end at the until keyword, but only after the condition. So, the condition may refer to local variables declared inside the loop block.

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

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

The break statement is 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.

The return and break statements may 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 may 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.

A.1.5.6 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),
                              tonumber(e3)
  if not (_var and _limit and _step) then error() end
  while (_step>0 and _var<=_limit)
        or (_step<=0 and _var>=_limit) do
    local var = _var
    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.

– var, _limit, and _step Are invisible variables. The names are here for explanatory purposes only.

– If the third expression (the step) is absent, then a step of 1 is used.

– Break may be used to exit a for loop.

– The loop variable var is local to the loop; its value shall not be used after the for ends or is broken. If the value of the loop variable var is needed, then assign it to another variable before breaking or exiting the loop.

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

```
stat ::= for namelist in explist1 do block end
```

```
namelist ::= Name {`,´ Name}
```

A for statement like

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

is equivalent to the code:

```
do
  local _f, _s, _var = explist
  while true do
    local var_1, ... , var_n = _f(_s, _var)
    _var = var_1
    if _var == 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, _s, And _var are invisible variables. The names are here for explanatory purposes only.

– Break may be used to exit a for loop.

– The loop variables var_i are local to the loop; their values shall not be used after the for ends. If these values are needed, then assign them to other variables before breaking or exiting the loop.

A.1.5.7 Function calls as statements

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

```
stat ::= functioncall
```

In this case, all returned values are thrown away. Function calls are explained in clause A.1.6.9.

A.1.5.8 Local declarations

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

```
stat ::= local namelist [`=´ explist1]
```

If present, an initial assignment has the same semantics of a multiple assignment (see clause A.1.5.4). Otherwise, all variables are initialized with nil.

A chunk is also a block (see clause A.1.5.2), and so local variables may be declared in a chunk outside any explicit block. The scope of such local variables extends until the end of the chunk.

The visibility rules for local variables are explained inclause A.1.7.

A.1.6 Expressions

A.1.6.1 Basic expressions

The basic expressions in Lua are the following:

```
exp ::= prefixexp
exp ::= nil  |  false  |  true
exp ::= Number
exp ::= String
exp ::= function
exp ::= tableconstructor
exp ::= `...´
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var  |  functioncall  |  `(´ exp `)´
```

Numbers and literal strings are explained in clause A.1.2; variables are explained in clause A.1.4; function definitions are explained in clause A.1.6.10; function calls are explained in clause A.1.6.9; table constructors are explained in clause A.1.6.8. Vararg expressions, denoted by three dots (`...´), may only be used inside vararg functions; they are explained in clause A.1.6.10.

Binary operators comprise arithmetic operators (see clause A.1.6.2), relational operators (see clause A.1.6.3), and logical operators (see clause A.1.6.4). Unary operators comprise the unary minus (see clause A.1.6.2), the unary not (see clause A.1.6.4), and the unary length operator (see clause A.1.6.6).

Both function calls and vararg expressions may result in multiple values. If the expression is used as a statement (see clause A.1.5.7) (only possible for function calls), then its return list is adjusted to zero elements, thus discarding all returned values. If the expression is used inside another expression or in the middle of a list of expressions, then its result list is adjusted to one element, thus discarding all values except the first one. If the expression is used as the last element of a list of expressions, then no adjustment is made, unless the call is enclosed in parentheses.

Table A.2 lists some examples.

Table A.2

`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 (c gets nil)
`a,b = ...`	-- a gets the first vararg parameter, b gets
	-- the second (both a and b may get nil if
	-- there is no corresponding vararg parameter)
`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 ...`	-- returns all received vararg parameters
`return x,y,f()`	-- returns x, y, and all values returned by f()
`{f()}`	-- creates a list with all values returned by f()
`{...}`	-- creates a list with all vararg parameters
`{f(), nil}`	-- f() is adjusted to 1 result

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.)

A.1.6.2 Arithmetic operators

Lua supports the usual arithmetic operators: the binary + (addition), - (subtraction), * (multiplication), / (division), % (modulo), and ^ (exponentiation); and unary - (negation). If the operands are numbers, or strings that may be converted to numbers (see clause A.1.3.2), then all operations have the usual meaning. Exponentiation works for any exponent. For instance, x^(-0.5) computes the inverse of the square root of x. Modulus is defined as

```
a % b == a - math.floor(a/b)*b
```

That is, it is the remainder of a division that rounds the quotient towards minus infinity.

A.1.6.3 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 a new object (a table, userdata, thread, or function) is created, this new object is different from any previously existing object.

The way that Lua compares tables and userdata may be changed by using the "eq" metamethod (see clause A.1.9).

The conversion rules of clause A.1.3.2 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 clause A.1.9).

A.1.6.4 Logical operators

The logical operators in Lua are

```
and   or    not
```

Like the control structures (see clause A.1.5.5), all logical operators consider both false and nil as false and anything else as true.

The negation operator does not always return 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; i.e., the second operand is evaluated only if necessary. Here are some examples:

```
10 or 20            --> 10
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
```

(Here and in the sequel, `-->´ indicates the result of the preceding expression.)

A.1.6.5 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 clause A.1.3.2. Otherwise, the "concat" metamethod is called (see clause A.1.9).

A.1.6.6 The length operator

The length operator is denoted by the unary operator #. The length of a string is its number of bytes (i.e., the usual meaning of string length when each character is 1 byte).

The length of a table t is defined to be any integer index n such that t[n] is not nil and t[n+1] is nil; moreover, if t[1] is nil, n may be zero. For a regular array, with non-nil values from 1 to a given n, its length is exactly that n, the index of its last value. If the array has "holes" (i.e., nil values between other non-nil values), then #t may be any of the indices that directly precedes a nil value (i.e., it may consider any such nil value as the end of the array).

A.1.6.7 Precedence

Operator precedence in Lua follows the following scheme, from lower to higher priority:

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

As usual, parentheses may be used to change the precedences of an expression. The concatenation (`..´) and exponentiation (`^´) operators are right associative. All other binary operators are left associative.

A.1.6.8 Table constructors

Table constructors are expressions that create tables. Every time a constructor is evaluated, a new table is created. Constructors may 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 t = {}
  t[f(1)] = g
  t[1] = "x"         -- 1st exp
  t[2] = "y"         -- 2nd exp
  t.x = 1            -- t["x"] = 1
  t[3] = f(x)        -- 3rd exp
  t[30] = 23
  t[4] = 45          -- 4th exp
  a = t
end
```

If the last field in the list has the form exp and the expression is a function call or a vararg expression, then all values returned by that expression enter the list consecutively (see clause A.1.6.9). To avoid this, enclose the function call (or the vararg expression) in parentheses (see clause A.1.6).

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

A.1.6.9 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, the prefixexp "call" metamethod is called, having as first parameter the value of prefixexp, followed by the original call arguments (see clause A.1.9).

The form

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

may 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 ::= String
```

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

As an exception to the free-format syntax of Lua, a line break shall not be put before the `(´ in a function call. That restriction avoids some ambiguities in the language. If one writes

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

Lua would see that as a single statement, a = f(g).x(a). So, if two statements are desired, a semi-colon must be added between them. If it is desired to call f, the line break before (g) must be removed.

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 may 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 return exactly the returns of the called function. So, none of the following examples are 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
```

A.1.6.10 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
```

not this:

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

(This only makes a difference when the body of the function contains references to f.)

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 [`,´ `...´]  |  `...´
```

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 and supplies them to the function through a vararg expression, which is also written as three dots. The value of this expression is a list of all actual extra arguments, similar to a function with multiple results. If a vararg expression is used inside another expression or in the middle of a list of expressions, then its return list is adjusted to one element. If the expression is used as the last element of a list of expressions, then no adjustment is made (unless the call is enclosed in parentheses).

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, the following mapping from arguments to parameters and to the vararg expression is obtained:

```
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, ... -->  (nothing)
g(3, 4)         a=3, b=4,   ... -->  (nothing)
g(3, 4, 5, 8)   a=3, b=4,   ... -->  5  8
g(5, r())       a=5, b=1,   ... -->  2  3
```

Results are returned using the return statement (see clauseA.1.5.5). 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, i.e., 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
```

A.1.7 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. Consider the following example:

```
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 may be freely accessed by functions defined inside their scope. 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 10 closures (i.e., 10 instances of the anonymous function). Each of these closures uses a different y variable, while all of them share the same x.

A.1.8 Error handling

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

Lua code may explicitly generate an error by calling the error function. The pcall function may be used to catch errors in Lua.

A.1.9 Metatables

Every value in Lua may have a metatable. This metatable is an ordinary Lua table that defines the behaviour of the original value under certain special operations. Several aspects of the behaviour of operations over a value may be changed by setting specific fields in its metatable. For instance, when a non-numeric value 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.

The keys are called 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.

One may query the metatable of any value through the getmetatable function.

The metatable of tables may be replaced through the setmetatable function. The metatable of other types from Lua shall not be changed (except using the debug library); instead, the C API must be used for that.

Tables and userdata have individual metatables (although multiple tables and userdata may share a same table as their metatable); values of all other types share one single metatable per type. So, there is one single metatable for all numbers, and for all strings, etc.

A metatable may control how an object behaves in arithmetic operations, order comparisons, concatenation, length operation, and indexing. A metatable may 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 value, it checks whether that value 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 in Lua shown in this clause is only illustrative; the real behaviour 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 9.2. In particular, to retrieve the metamethod of a given object, the following expression is used

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

This should be read as

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

i.e., 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 behaviour 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 behaviour
      error("...")
    end
  end
end
```

– "sub": the - operation. Behaviour similar to the "add" operation.

– "mul": the * operation. Behaviour similar to the "add" operation.

– "div": the / operation. Behaviour similar to the "add" operation.

– "mod": the % operation. Behaviour similar to the "add" operation, with the operation o1 - floor(o1/o2)*o2 as the primitive operation.

– "pow": the ^ (exponentiation) operation. Behaviour similar to the "add" operation, with the function pow (from the C math library) as the primitive operation.

– "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
      return h(op)
    else  -- no handler available: default behaviour
      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
```

– "len": the # operation.

```
function len_event (op)
  if type(op) == "string" then
    return strlen(op)         -- primitive string length
  elseif type(op) == "table" then
    return #op                -- primitive table length
  else
    local h = metatable(op).__len
    if h then
      -- call the handler with the operand
      return h(op)
    else  -- no handler available: default behaviour
      error("...")
    end
  end
end
```

See clause A.1.6.6 for a description of the length of a table.

– "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
```

`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
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
end
```

– "call": called when Lua calls a value.

```
function function_event (func, ...)
  if type(func) == "function" then
    return func(...)   -- primitive call
  else
```

```
    local h = metatable(func).__call
    if h then
      return h(func, ...)
    else
      error("...")
    end
  end
end
```

A.1.10 Environments

Besides metatables, objects of types thread, function and userdata have another table associated with them, called their environment. Like metatables, environments are regular tables and multiple objects may share the same environment.

Environments associated with userdata have no meaning for Lua. It is only a feature for programmers to associate a table to a userdata.

Environments associated with threads are called global environments. They are used as the default environment for threads and non-nested functions created by that thread (through loadfile, loadstring or load) and may be directly accessed by C code (see clause A.2.4).

Environments associated with C functions may be directly accessed by C code (see clause A.2.4). They are used as the default environment for other C functions created by that function.

Environments associated with Lua functions are used to resolve all accesses to global variables within that function (see clause A.1.4). They are used as the default environment for other Lua functions created by that function.

The environment of a Lua function or the running thread may be changed by calling setfenv. The environment of a Lua function or of the running thread may be obtained by calling getfenv. To manipulate the environment of other objects (userdata, C functions, other threads), the C API must be used.

A.1.11 Garbage collection

A.1.11.1 Basic concepts

Lua performs automatic memory management. That means that concern is necessary neither about allocating memory for new objects, nor about 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 (i.e., 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 implements an incremental mark-and-sweep collector. It uses two numbers to control its garbage-collection cycles: the garbage-collector pause and the garbage-collector step multiplier.

The garbage-collector pause controls how long the collector waits before starting a new cycle. Larger values make the collector less aggressive. Values smaller than 1 mean the collector will not wait to start a new cycle. A value of 2 means that the collector waits more or less to double the total memory in use before starting a new cycle.

The step multiplier controls the relative speed of the collector relative to memory allocation. Larger values make the collector more aggressive but also increases the size of each incremental step. Values smaller than 1 make the collector too slow and can result in the collector never finishing a cycle. The default, 2, means that the collector runs at twice the speed of memory allocation.

Those numbers calling lua_gc in C or collectgarbage in Lua may be changed. Both get as arguments percentage points (so an argument 100 means a real value of 1). With those functions direct control of the collector (e.g., stop and restart it) may also be obtained.

A.1.11.2 Garbage-collection metamethods

Using the C API, garbage-collector metamethods may be set for userdata (see clause A.1.9). These metamethods are also called finalizers. Finalizers allow coordination of Lua's garbage collection with external resource management (such as closing files, network or database connections or freeing the developer's own memory).

Garbage 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.

A.1.11.3 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 may 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 a table is used as a metatable, the value of its field __mode should not be changed. Otherwise, the weak behaviour of the tables controlled by this metatable is undefined.

A.1.12 Coroutines

Lua supports coroutines, also called 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.

A coroutine is created 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 coroutine.resume, passing as its first argument the thread returned by coroutine.create, is first called, the coroutine starts its execution, at the first line of its main function. Extra arguments passed to coroutine.resume are passed on to the coroutine main function. After the coroutine starts running, it runs until it terminates or yields.

A coroutine may 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 (i.e., 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 the same coroutine is resumed, 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, just 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 coroutine.resume. coroutine.wrap Returns all the values returned by coroutine.resume, except the first one (the boolean error code). Unlike coroutine.resume, coroutine.wrap does not catch errors; any error is propagated to the caller.

As an example, consider the next code:

```
function foo (a)
  print("foo", a)
  return coroutine.yield(2*a)
end
co = coroutine.create(function (a,b)
      print("co-body", a, b)
      local r = foo(a+1)
      print("co-body", r)
      local r, s = coroutine.yield(a+b, a-b)
      print("co-body", r, s)
      return b, "end"
end)
print("main", coroutine.resume(co, 1, 10))
print("main", coroutine.resume(co, "r"))
print("main", coroutine.resume(co, "x", "y"))
print("main", coroutine.resume(co, "x", "y"))
```

When it is run, 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
```

A.2 The application program interface

A.2.1 Basic concepts

All C API functions and related types and constants are declared in the header file lua.h.

Even when the term "function" is used, 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 any hidden side-effects.

As in most C libraries, the Lua API functions do not check their arguments for validity or consistency. However, this behaviour may be changed by compiling Lua with a proper definition for the macro luai_apicheck, in file luaconf.h [b-luaconf.h].

A.2.2 The stack

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 lua_CFunction).

For convenience, most query operations in the API do not follow a strict stack discipline. Instead, they may 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 relative to the top of the stack. More specifically, if the stack has n elements, then index 1 represents the first element (i.e., the element that was pushed on to the stack first) and index n represents the last element; index –1 also represents the last element (i.e., the element at the top) and index -n represents the first element. It can be said that an index is valid if it lies between 1 and the stack top (i.e., if `1 <= abs(index) <= top`).

A.2.3 Stack size

When interacting with Lua API, the developer is responsible for ensuring consistency, in particular, for controlling stack overflow. The function lua_checkstack may be used to grow the stack size.

Whenever Lua calls C, it ensures that at least LUA_MINSTACK stack positions are available. LUA_MINSTACK is defined as 20, so that usually there is no need to worry about stack space unless the code has loops pushing elements on to the stack.

Most query functions accept as indices any value inside the available stack space, i.e., indices up to the maximum stack size set through lua_checkstack. Such indices are called acceptable indices. More formally, an acceptable index is defined as follows:

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

Note that 0 is never an acceptable index.

A.2.4 Pseudo-indices

Unless otherwise noted, any function that accepts valid indices may also be called with pseudo-indices, which represent some Lua values that are accessible to C code, but which are not in the stack. Pseudo-indices are used to access the thread environment, the function environment, the registry and the upvalues of a C function (see clause A.2.5).

The thread environment (where global variables live) is always at pseudo-index LUA_GLOBALSINDEX. The environment of the running C function is always at pseudo-index LUA_ENVIRONINDEX.

To access and change the value of global variables, regular table operations may be used over an environment table. For instance, to access the value of a global variable, do

```
lua_getfield(L, LUA_GLOBALSINDEX, varname);
```

A.2.5 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 called upvalues and are accessible to the function whenever it is called (see lua_pushcclosure).

Whenever a C function is called, its upvalues are located at specific pseudo-indices. Those pseudo-indices are produced by the 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.

A.2.6 Registry

Lua provides a registry, a pre-defined table that may be used by any C code to store whatever Lua value it needs to store. This table is always located at pseudo-index LUA_REGISTRYINDEX. Any C library may store data in this table, but it should take care to choose keys different from those

used by other libraries, to avoid collisions. Typically, the developer should use as key a string containing the library name or a light userdata with the address of a C object in the code.

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

A.2.7 Error handling in C

Internally, Lua uses the C longjmp facility to handle errors. (The developer may also choose to use exceptions if C++ is used; see file luaconf.h [b-luaconf.h].) When Lua faces any error (such as memory allocation errors, type errors, syntax errors, and runtime errors) it raises an error; i.e., 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.

Almost any function in the API may raise an error, e.g., due to a memory allocation error. The following functions run in protected mode (i.e., they create a protected environment to run), so they never raise an error: lua_newstate, lua_close, lua_load, lua_pcall and lua_cpcall.

Inside a C function an error may be raised by calling lua_error.

A.2.8 Functions and types

This clause lists all functions and types from the C API in alphabetical order.

---

lua_Alloc

```
typedef void * (*lua_Alloc) (void *ud, void *ptr, size_t osize,
size_t nsize);
```

The type of memory allocation function used by Lua states. The allocator function must provide a functionality similar to realloc, but not exactly the same. Its arguments are ud, an opaque pointer passed to lua_newstate; ptr, a pointer to the block being allocated/reallocated/freed; osize, the original size of the block; nsize, the new size of the block. ptr is NULL if and only if osize is zero. When nsize is zero, the allocator must return NULL; if osize is not zero, it should free the block pointed by ptr. When nsize is not zero, the allocator returns NULL if and only if it cannot fill the request. When nsize is not zero and osize is zero, the allocator should behave like malloc. When nsize and osize are not zero, the allocator behaves like realloc. Lua assumes that the allocator never fails when osize >= nsize.

Here is a simple implementation for the allocator function. It is used in the auxiliary library by lua_newstate.

```
static void *l_alloc (void *ud, void *ptr, size_t osize, size_t nsize) {
  (void)ud;       /* not used */
  (void)osize;   /* not used */
  if (nsize == 0) {
    free(ptr);  /* ANSI requires that free(NULL) has no effect */
    return NULL;
  }
  else
    /* ANSI requires that realloc(NULL, size) == malloc(size) */
    return realloc(ptr, nsize);
}
```

---

lua_atpanic

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

sets a new panic function and returns the old one.

If an error happens outside any protected environment, Lua calls a panic function and then calls exit(EXIT_FAILURE), thus exiting the host application. The panic function can avoid this exit by never returning (e.g., doing a long jump).

The panic function may access the error message at the top of the stack.

---

lua_call

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

calls a function.

To call a function the following protocol must be used: First, the function to be called is pushed on to the stack; then, the arguments to the function are pushed in direct order; i.e., the first argument is pushed first. Finally, lua_call is called; nargs is the number of arguments pushed on to the stack. All arguments and the function value are popped from the stack when the function is called. The function results are pushed on to the stack when the function returns. The number of results is 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 on to the stack in direct order (the first result is pushed first), so that after the call, the last result is on the top of the stack.

Any error inside the called function is propagated upwards (with a longjmp).

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

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

Here it is in C:

```
  lua_getfield(L, LUA_GLOBALSINDEX, "f");  /* function to be called */
  lua_pushstring(L, "how");       /* 1st argument */
  lua_getfield(L, LUA_GLOBALSINDEX, "t");  /* table to be indexed */
  lua_getfield(L, -1, "x");           /* push result of t.x (2nd arg) */
  lua_remove(L, -2);                  /* remove `t' from the stack */
  lua_pushinteger(L, 14);             /* 3rd argument */
  lua_call(L, 3, 1);                      /* call function with 3 arguments and
1 result */
  lua_setfield(L, LUA_GLOBALSINDEX, "a");  /* set global variable `a' */
```

Note that the code above is "balanced": at its end, the stack is back to its original configuration. This is considered good programming practice.

---

lua_CFunction

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

defines the type for C functions.

In order to communicate properly with Lua, a C function must use 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, lua_gettop(L) returns the number of arguments received by the function. The first argument (if any) is at index 1 and its last argument is at index lua_gettop(L). To return values to Lua, a C function just pushes them on to 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 discarded by Lua. Like a Lua function, a C function called by Lua may also return many results.

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

```
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 `average'");
    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 */
}
```

---

lua_checkstack

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

ensures that there are at least extra free stack slots in the stack.

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.

---

lua_close

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

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, this function does not need to be called, 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.

---

lua_concat

```
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 (i.e., the function does nothing); if n is 0, the result is the empty string. Concatenation is done following the usual semantics of Lua (see clause A.1.6.5).

---

lua_cpcall

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

calls the C function 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, plus the error object on the top of the stack; otherwise, it returns zero, and does not change the stack. All values returned by func are discarded.

---

lua_createtable

```
void lua_createtable (lua_State *L, int narr, int nrec);
```

creates a new empty table and pushes it on to the stack.

The new table has space pre-allocated for narr array elements and nrec non-array elements. This pre-allocation is useful when the exact number of elements the table will have is known. Otherwise, the function lua_newtable may be used.

lua_dump

```
int lua_dump (lua_State *L, lua_Writer writer, void *data);
```

dumps a function as a binary chunk.

This function receives a Lua function on the top of the stack and produces a binary chunk that, if loaded again, results in a function equivalent to the one dumped. As it produces parts of the chunk, lua_dump calls function writer (see lua_Writer) with the given data to write them.

The value returned is the error code returned by the last call to the writer; 0 means no errors.

This function does not pop the function from the stack.

lua_equal

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

returns 1 if the two values in acceptable indices index1 and index2 are equal, following the semantics of the Lua == operator (i.e., may call metamethods).

Otherwise it returns 0. It also returns 0 if any of the indices are non-valid.

lua_error

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

generates a Lua error.

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

lua_gc

```
int lua_gc (lua_State *L, int what, int data);
```

controls the garbage collector.

This function performs several tasks, according to the value of the parameter what.

– LUA_GCSTOP--- stops the garbage collector.

– LUA_GCRESTART--- restarts the garbage collector.

– LUA_GCCOLLECT--- performs a full garbage-collection cycle.

– LUA_GCCOUNT--- returns the current amount of memory (in Kbytes) in use by Lua.

– LUA_GCCOUNTB--- returns the remainder of dividing the current amount of bytes of memory in use by Lua by 1024.

– LUA_GCSTEP--- performs an incremental step of garbage collection. The step "size" is controlled by data (larger values mean more steps) in a non-specified way. If the developer wants to control the step size, the value of data must be tuned experimentally. The function returns 1 if that step finished a garbage-collection cycle.

– LUA_GCSETPAUSE--- sets data/100 as the new value for the pause of the collector (see clause A.1.11). The function returns the previous value of the pause.

– LUA_GCSETSTEPMUL--- sets arg/100 as the new value for the step multiplier of the collector (see clause A.1.11). The function returns the previous value of the step multiplier.

lua_getallocf

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
```

returns the memory allocator function of a given state.

If ud is not NULL, Lua stores in *ud the opaque pointer passed to lua_newstate.

---

lua_getfenv

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

pushes the environment table of the value at the given index on to the stack.

---

lua_getfield

```
void lua_getfield (lua_State *L, int index, const char *k);
```

pushes the value t[k], where t is the value at the given valid index index on to the stack. As in Lua, this function may trigger a metamethod for the "index" event (see clause A.1.9).

---

lua_getglobal

```
void lua_getglobal (lua_State *L, const char *name);
```

pushes the value of the global name on to the stack.

It is defined as a macro:

```
#define lua_getglobal(L,s)  lua_getfield(L, LUA_GLOBALSINDEX, s)
```

---

lua_getmetatable

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

pushes the metatable of the value at the given acceptable index on to the stack. If the index is not valid, or if the value does not have a metatable, the function returns 0 and pushes nothing on the stack.

---

lua_gettable

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

pushes the value t[k], where t is the value at the given valid index index and k is the value at the top of the stack, on to the stack.

This function pops the key from the stack (putting the resulting value in its place). As in Lua, this function may trigger a metamethod for the "index" event (see clause A.1.9).

---

lua_gettop

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

returns the index of the top element in the stack.

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

---

lua_insert

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

moves the top element into the given valid index, shifting up the elements above that position to open space.

This function shall not be called with a pseudo-index, because a pseudo-index is not an actual stack position.

---

lua_Integer

```
typedef ptrdiff_t lua_Integer;
```

is the type used by the Lua API to represent integral values.

By default it is a ptrdiff_t, which is usually the largest integral type the machine handles "comfortably".

---

lua_isboolean

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

returns 1 if the value at the given acceptable index has type boolean and 0 otherwise.

---

lua_iscfunction

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

returns 1 if the value at the given acceptable index is a C function and 0 otherwise.

---

lua_isfunction

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

returns 1 if the value at the given acceptable index is a function (either C or Lua) and 0 otherwise.

---

lua_islightuserdata

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

returns 1 if the value at the given acceptable index is a light userdata and 0 otherwise.

---

lua_isnil

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

returns 1 if the value at the given acceptable index is nil and 0 otherwise.

---

lua_isnumber

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

returns 1 if the value at the given acceptable index is a number or a string convertible to a number and 0 otherwise.

---

lua_isstring

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

returns 1 if the value at the given acceptable index is a string or a number (which is always convertible to a string) and 0 otherwise.

---

lua_istable

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

returns 1 if the value at the given acceptable index is a table and 0 otherwise.

---

lua_isthread

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

returns 1 if the value at the given acceptable index is a thread and 0 otherwise.

---

lua_isuserdata

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

returns 1 if the value at the given acceptable index is a userdata (either full or light) and 0 otherwise.

---

lua_lessthan

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

returns 1 if the value at acceptable index index1 is smaller than the value at acceptable index index2, following the semantics of the Lua < operator (i.e., may call metamethods).

Otherwise this function returns 0. It also returns 0 if any of the indices is not-valid.

---

lua_load

```
int lua_load (lua_State *L, lua_Reader reader, void *data, const char
*chunkname);
```

loads a Lua chunk.

If there are no errors, lua_load pushes the compiled chunk as a Lua function to the top of the stack. Otherwise, it pushes an error message. The return values of lua_load are:

– 0 --- no errors;

– LUA_ERRSYNTAX --- syntax error during pre-compilation;

– LUA_ERRMEM --- memory allocation error.

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 (see lua_Reader). The data argument is an opaque value passed to the reader function.

The chunkname argument gives a name to the chunk, which is used for error messages and in debug information (see clause A.2.9).

---

lua_newstate

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```

creates a new, independent state.

This function returns NULL if cannot create the state (due to lack of memory). The argument f is the allocator function; Lua does all memory allocation for that state through that function. The second argument, ud, is an opaque pointer that Lua simply passes to the allocator in every call.

---

lua_newtable

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

creates a new empty table and pushes it on to the stack.

Equivalent to lua_createtable(L, 0, 0).

---

lua_newthread

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

creates a new thread, pushes it on to the stack and returns a pointer to a lua_State that represents this new thread.

The new state returned by this function shares all global objects (such as tables) with the original state, but has an independent execution stack.

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

---

lua_newuserdata

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

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

Userdata represents C values in Lua. A full userdata represents a block of memory. It is an object (like a table): the developer must create it, it may have its own metatable, and the developer may detect when it is being collected. A full userdata is only equal to itself (under raw equality).

When Lua collects a full userdata with a gc metamethod, Lua calls the metamethod and marks the userdata as finalized. When that userdata is collected again, then Lua frees its corresponding memory.

---

lua_next

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

pops a key from the stack, and pushes a key-value pair from the table at the given index (the "next" pair after the given key).

If there are no more elements in the table, then lua_next returns 0 (and pushes nothing).

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_type(L, -1)));
  lua_pop(L, 1);  /* removes `value'; keeps `key' for next iteration */
}
```

While traversing a table, do not call lua_tolstring directly on a key, unless it is known that the key is actually a string. Recall that lua_tolstring changes the value at the given index; this confuses the next call to lua_next.

---

lua_Number

```
typedef double lua_Number;
```

is the type of numbers in Lua.

By default, it is double, but that may be changed in luaconf.h [b-luaconf.h].

Through the configuration file, the developer may change Lua to operate with another type for numbers (e.g., float or long).

---

lua_objlen

```
size_t lua_objlen (lua_State *L, int index);
```

returns the "length" of the value at the given acceptable index: for strings, this is the string length; for tables, this is the result of the length operator (`#´); for userdata, this is the size of the block of memory allocated for the userdata; for other values, it is 0.

lua_pcall

```
lua_pcall (lua_State *L, int nargs, int nresults, int errfunc);
```

calls a function in protected mode.

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 on to 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 on the stack is exactly the original error message. Otherwise, errfunc is the stack index of an error handler function. (In the current implementation, that index shall not be a pseudo-index.) In the case of runtime errors, that function will be called with the error message and its return value will be the message returned on the stack 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 the 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.

lua_pop

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

pops n elements from the stack.

lua_pushboolean

```
void lua_pushboolean (lua_State *L, int b);
```

pushes a boolean value with value b on to the stack.

lua_pushcclosure

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

pushes a new C closure on to the stack.

When a C function is created, it is possible to associate some values with it, thus creating a C closure (see clause A.2.5); 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 on to the stack (when there are multiple values, the first value is pushed first). Then lua_pushcclosure is called to create and push the C function on to the stack, with the argument n indicating how many values should be associated with the function. lua_pushcclosure Also pops these values from the stack.

lua_pushcfunction

```
void lua_pushcfunction (lua_State *L, lua_CFunction f);
```

pushes a C function on to the stack.

This function receives a pointer to a C function and pushes a Lua value of type function on to the stack that, when called, invokes the corresponding C function.

Any function to be registered in Lua must follow the correct protocol to receive its parameters and return its results (see lua_CFunction).

The call lua_pushcfunction(L, f) is equivalent to lua_pushcclosure(L, f, 0).

---

lua_pushfstring

```
const char *lua_pushfstring (lua_State *L, const char *fmt, ...);
```

pushes a formatted string on to the stack and returns a pointer to that string.

It is similar to the C function sprintf, but has some important differences.

– Allocation of space for the result is not necessary: 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 may only be `%%´ (inserts a `%´ in the string), `%s´ (inserts a zero-terminated string, with no size restrictions), `%f´ (inserts a lua_Number), `%p´ (inserts a pointer as an hexadecimal numeral), `%d´ (inserts an int) and `%c´ (inserts an int as a character).

---

lua_pushinteger

```
void lua_pushinteger (lua_State *L, lua_Integer n);
```

pushes a number with value n on to the stack.

---

lua_pushlightuserdata

```
void lua_pushlightuserdata (lua_State *L, void *p);
```

pushes a light userdata on to the stack.

Userdata represents C values in Lua. A light userdata represents a pointer. It is a value (like a number): the developer does 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.

---

lua_pushlstring

```
void lua_pushlstring (lua_State *L, const char *s, size_t len);
```

pushes the string pointed by s with size len on to the stack.

Lua makes (or reuses) an internal copy of the given string, so the memory at s may be freed or reused immediately after the function returns. The string may contain embedded zeros.

---

lua_pushnil

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

pushes a nil value on to the stack.

---

lua_pushnumber

```
void lua_pushnumber (lua_State *L, lua_Number n);
```

pushes a number with value n on to the stack.

---

lua_pushstring

```
void lua_pushstring (lua_State *L, const char *s);
```

pushes the zero-terminated string pointed by s on to the stack.

Lua makes (or reuses) an internal copy of the given string, so the memory at s may be freed or reused immediately after the function returns. The string shall not contain embedded zeros; it is assumed to end at the first zero.

---

lua_pushthread

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

pushes the thread represented by L on to the stack.

---

lua_pushvalue

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

pushes a copy of the element at the given valid index on to the stack.

---

lua_pushvfstring

```
const char *lua_pushvfstring (lua_State *L, const char *fmt,
                              va_list argp);
```

is equivalent to lua_pushfstring, except that it receives a va_list instead of a variable number of arguments.

---

lua_rawequal

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

returns 1 if the two values in acceptable indices index1 and index2 are primitively equal (i.e., without calling metamethods). Otherwise the function returns 0. It also returns 0 if any of the indices are non-valid.

---

lua_rawget

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

is similar to lua_gettable, but does a raw access (i.e., without metamethods).

---

lua_rawgeti

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

pushes the value t[n], where t is the value at the given valid index index, on to the stack. The access is raw; i.e., it does not invoke metamethods.

---

lua_rawset

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

is similar to lua_settable, but does a raw assignment (i.e., without metamethods).

---

lua_rawseti

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

does the equivalent of t[n] = v, where t is the value at the given valid index index and v is the value at the top of the stack.

This function pops the value from the stack. The assignment is raw; i.e., it does not invoke metamethods.

lua_Reader

```
typedef const char * (*lua_Reader)
                     (lua_State *L, void *data, size_t *size);
```

is used by lua_load.

Every time 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. The block must exist until the reader function is called again. To signal the end of the chunk, the reader must return NULL. The reader function may return pieces of any size greater than zero.

lua_register

```
void lua_register (lua_State *L, const char *name, lua_CFunction f);
```

sets the C function f as the new value of global name.

It is defined as a macro:

#define lua_register(L,n,f) (lua_pushcfunction(L, f), lua_setglobal(L, n))

lua_remove

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

removes the element at the given valid index, shifting down the elements above that position to fill the gap.

This function shall not be called with a pseudo-index, because a pseudo-index is not an actual stack position.

lua_replace

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

moves the top element into the given position (and pops it), without shifting any element (therefore replacing the value at the given position).

lua_resume

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

starts and resumes a coroutine in a given thread.

To start a coroutine, a new thread (see lua_newthread) is first created; then the main function plus any arguments is pushed on to its stack; then lua_resume is called, 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 LUA_YIELD if the coroutine yields, 0 if the coroutine finishes its execution without errors or an error code in the case of errors (see lua_pcall). In the case of errors, the stack is not unwound, so the debug API may be used over it. The error message is on the top of the stack. To restart a coroutine, only the values to be passed as results from yield are put on to its stack, and then lua_resume is called.

lua_setallocf

```
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);
```

changes the allocator function of a given state to f with user data ud.

lua_setfenv

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

pops a table from the stack and sets it as the new environment for the value at the given index.

If the value at the given index is not a function, not a thread and not a userdata, lua_setfenv returns 0. Otherwise it returns 1.

lua_setfield

```
void lua_setfield (lua_State *L, int index, const char *k);
```

does the equivalent to t[k] = v, where t is the value at the given valid index index and v is the value at the top of the stack.

This function pops the value from the stack. As in Lua, this function may trigger a metamethod for the "newindex" event (see clause A.1.9).

lua_setglobal

```
void lua_setglobal (lua_State *L, const char *name);
```

pops a value from the stack and sets it as the new value of global name.

It is defined as a macro:

```
#define lua_setglobal(L,s)   lua_setfield(L, LUA_GLOBALSINDEX, s)
```

lua_setmetatable

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

pops a table from the stack and sets it as the new metatable for the value at the given acceptable index.

lua_settable

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

does the equivalent to t[k] = v, where t is the value at the given valid index index, v is the value at the top of the stack and k is the value just below the top.

This function pops both the key and the value from the stack. As in Lua, this function may trigger a metamethod for the "newindex" event (see clause A.1.9).

lua_settop

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

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 the index is 0, then all stack elements are removed.

lua_State

```
typedef struct lua_State lua_State;
```

is an opaque structure that keeps the whole state of a Lua interpreter.

The Lua library is fully reentrant: it has no global variables. All information about a state is kept in this structure.

A pointer to this state must be passed as the first argument to every function in the library, except to lua_newstate, which creates a Lua state from scratch.

lua_status

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

returns the status of the thread L.

The status may be 0 for a normal thread, an error code if the thread finished its execution with an error or LUA_YIELD if the thread is suspended.

lua_toboolean

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

converts the Lua value at the given acceptable 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 the developer wants to accept only actual boolean values, use lua_isboolean to test the value's type.)

lua_tocfunction

```
lua_CFunction lua_tocfunction (lua_State *L, int index);
```

converts a value at the given acceptable index to a C function.

That value must be a C function; otherwise, returns NULL.

lua_tointeger

```
lua_Integer lua_tointeger (lua_State *L, int idx);
```

converts the Lua value at the given acceptable index to the signed integral type lua_Integer.

The Lua value must be a number or a string convertible to a number (see clause A.1.3.2); otherwise, lua_tointeger returns 0.

If the number is not an integer, it is truncated in some non-specified way.

lua_tolstring

```
const char *lua_tolstring (lua_State *L, int index, size_t *len);
```

converts the Lua value at the given acceptable index to a string (const char*).

If len is not NULL, it also sets *len with the string length. The Lua value must be a string or a number; otherwise, the function returns NULL. If the value is a number, then lua_tolstring also changes the actual value in the stack to a string. (This change confuses lua_next when lua_tolstring is applied to keys during a table traversal.)

lua_tolstring 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. Because Lua has garbage collection, there is no guarantee that the pointer returned by lua_tolstring will be valid after the corresponding value is removed from the stack.

lua_tonumber

```
lua_Number lua_tonumber (lua_State *L, int index);
```

converts the Lua value at the given acceptable index to a number (see lua_Number).

The Lua value must be a number or a string convertible to a number (see clause A.1.3.2); otherwise, lua_tonumber returns 0.

lua_topointer

```
const void *lua_topointer (lua_State *L, int index);
```

converts the value at the given acceptable index 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 return different pointers. There is no direct way to convert the pointer back to its original value.

Typically this function is used only for debug information.

lua_tostring

```
const char *lua_tostring (lua_State *L, int index);
```

is equivalent to lua_tolstring with len equal to NULL.

lua_tothread

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

converts the value at the given acceptable index to a Lua thread (represented as lua_State*).

This value must be a thread; otherwise, the function returns NULL.

lua_touserdata

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

If the value at the given acceptable index is a full userdata, returns its block address. If the value is a light userdata, returns its pointer. Otherwise, returns NULL.

lua_type

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

returns the type of the value in the given acceptable index or LUA_TNONE for a non-valid index (i.e., an index to an "empty" stack position).

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, and LUA_TLIGHTUSERDATA.

lua_typename

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

returns the name of the type encoded by the value tp, which must be one the values returned by lua_type.

lua_Writer

```
typedef int (*lua_Writer) (lua_State *L, const void* p, size_t sz, void* ud);
```

is the writer function used by lua_dump.

Every time it produces another piece of chunk, lua_dump calls the writer, passing along the buffer to be written (p), its size (sz) and the data parameter supplied to lua_dump.

The writer returns an error code: 0 means no errors; any other value means an error and stops lua_dump from calling the writer again.

lua_xmove

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

exchanges values between different threads of the same global state.

This function pops n values from the stack from, and pushes them on to the stack to.

lua_yield

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

yields a coroutine.

This function should 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.

A.2.9 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.

lua_Debug

```
typedef struct lua_Debug {
  int event;
  const char *name;       /* (n) */
  const char *namewhat;   /* (n) */
  const char *what;       /* (S) */
  const char *source;          /* (S) */
  int currentline;        /* (l) */
  int nups;               /* (u) number of upvalues */
  int linedefined;        /* (S) */
  int lastlinedefined;         /* (S) */
  char short_src[LUA_IDSIZE]; /* (S) */
  /* private part */
  ...
} lua_Debug;
```

is a structure used to carry different pieces of information about an active function.

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 lua_getinfo.

The fields of lua_Debug have the following meanings.

– 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.

– lastlinedefined --- The line number where the definition of the function ends.

– what --- The string "Lua" if the function is a Lua function, "C" if it is a C function, "main" if it is the main part of a chunk and "tail" if it was a function that did a tail call. In the latter case, Lua has no other information about the 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 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 may be "global", "local", "method", "field", "upvalue" 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.

---

lua_gethook

```
lua_Hook lua_gethook (lua_State *L);
```

returns the current hook function.

---

lua_gethookcount

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

returns the current hook count.

---

lua_gethookmask

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

returns the current hook mask.

---

lua_getinfo

```
int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);
```

fills the fields of lua_Debug with useful information.

This function returns 0 on error (e.g., 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: `S´ fills in the fields source, linedefined, lastlinedefined, and what; `l´ fills in the field currentline, etc. Moreover, `f´ pushes the function that is running at the given level on to the stack.

To get information about a function that is not active (i.e., not in the stack), it is pushed on to the stack and the what string starts with the character `>´. For instance, to know in which line a function f was defined, the following code may be written:

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

---

lua_getlocal

```
const char *lua_getlocal (lua_State *L, const lua_Debug *ar, int n);
```

gets information about a local variable of a given activation record.

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 lua_Hook). The index n selects which local variable to inspect

(1 is the first parameter or active local variable, and so on, until the last active local variable). lua_getlocal pushes the variable's value on to the stack and returns its name.

Variable names starting with `(´ (open parenthesis) represent internal variables (loop control variables, temporaries and C function locals.

This function returns NULL (and pushes nothing) when the index is greater than the number of active local variables.

---

lua_getstack

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

gets information about the interpreter runtime stack.

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.

---

lua_getupvalue

```
const char *lua_getupvalue (lua_State *L, int funcindex, int n);
```

gets information about a closure's upvalue.

(For Lua functions, upvalues are the external local variables that the function uses and that consequently are included in its closure.) lua_getupvalue Gets the index n of an upvalue, pushes the upvalue value on to the stack, and returns its name. funcindex Points to the closure in the stack. (Upvalues have no particular order, as they are active through the whole function. So, they are numbered in an arbitrary order.)

This function returns NULL (and pushes nothing) when the index is greater than the number of upvalues. For C functions, this function uses the empty string "" as a name for all upvalues.

---

lua_Hook

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

is a type for debugging hook functions.

Whenever a hook is called, its ar argument has its field event set to the specific event that triggered the hook. Lua identifies these events with the following constants: LUA_HOOKCALL, LUA_HOOKRET, LUA_HOOKTAILRET, LUA_HOOKLINE and LUA_HOOKCOUNT. 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 Lua back to execute a function or a chunk, that execution occurs without any calls to hooks.

---

lua_sethook

```
int lua_sethook (lua_State *L, lua_Hook func, int mask,  int count);
```

sets the debugging hook function.

func Is the hook function. mask Specifies on which events the hook will be called: It is formed by a bitwise or 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 follows.

– The call hook is called when the interpreter calls a function. The hook is called just after Lua enters the new function, before the function gets its arguments.
– The return hook is called when the interpreter returns from a function. The hook is called just before Lua leaves the function. The developer has no access to the values to be returned by 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.

---

lua_setlocal

```
const char *lua_setlocal (lua_State *L, const lua_Debug *ar, int n);
```

sets the value of a local variable of a given activation record.

Parameters ar and n are as in lua_getlocal (see lua_getlocal). lua_setlocal Assigns the value at the top of the stack to the variable and returns its name. It also pops the value from the stack.

This function returns NULL (and pops nothing) when the index is greater than the number of active local variables.

---

lua_setupvalue

```
const char *lua_setupvalue (lua_State *L, int funcindex, int n);
```

sets the value of a closure's upvalue.

Parameters funcindex and n are as in lua_getupvalue (see lua_getupvalue). It assigns the value at the top of the stack to the upvalue and returns its name. It also pops the value from the stack.

This function returns NULL (and pops nothing) when the index is greater than the number of upvalues.

A.3 The auxiliary library

A.3.1 Basic concepts

The auxiliary library provides several convenient functions to interface C with Lua. While the basic API provides the primitive functions for all interactions between C and Lua, the auxiliary library provides higher-level functions for some common tasks.

All functions from the auxiliary library are defined in header file lauxlib.h and have a prefix luaL_.

All functions in the auxiliary library are built on top of the basic API, and so they provide nothing that cannot be done with that API.

Several functions in the auxiliary library are used to check C function arguments. Their names are always luaL_check* or luaL_opt*. All of these functions raise an error if the check is not satisfied. Because the error message is formatted for arguments (e.g., "bad argument #1"), these functions should not be used for other stack values.

A.3.2 Functions and types

This clause list alls functions and types from the auxiliary library in alphabetical order.

luaL_addchar

```
void luaL_addchar (luaL_Buffer B, char c);
```

adds the character c to the buffer B (see luaL_Buffer).

luaL_addlstring

```
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
```

adds the string pointed by s with length l to the buffer B (see luaL_Buffer). The string may contain embedded zeros.

luaL_addsize

```
void luaL_addsize (luaL_Buffer B, size_t n);
```

adds a string of length n previously copied to the buffer area (see luaL_prepbuffer) to the buffer B (see luaL_Buffer).

luaL_addstring

```
void luaL_addstring (luaL_Buffer *B, const char *s);
```

adds the zero-terminated string pointed by s to the buffer B (see luaL_Buffer). The string shall not contain embedded zeros.

luaL_addvalue

```
void luaL_addvalue (luaL_Buffer *B);
```

adds the value at the top of the stack to the buffer B (see luaL_Buffer) and pops the value.

This is the only function on string buffers that may (and must) be called with an extra element on the stack, which is the value to be added to the buffer.

luaL_argcheck

```
void luaL_argcheck (lua_State *L, int cond, int numarg, const char
*extramsg);
```

checks whether cond is true.

If not, ths function raises an error with the message "bad argument #<numarg> to <func> (<extramsg>)", where func is retrieved from the call stack.

luaL_argerror

```
int luaL_argerror (lua_State *L, int numarg, const char *extramsg);
```

raises an error with the message "bad argument #<numarg> to <func> (<extramsg>)", where func is retrieved from the call stack.

This function never returns, but it is an idiom to use it as return luaL_argerror ... in C functions.

luaL_Buffer

```
typedef struct luaL_Buffer luaL_Buffer;
```

is a type for a string buffer.

A string buffer allows C code to build Lua strings piecemeal. Its pattern of use is as follows.

– First a variable b of type luaL_Buffer is declared

– Then it is initialized with a call luaL_buffinit(L, &b).

– Then string pieces are added to the buffer calling any of the luaL_add* functions.

– Finally, luaL_pushresult(&b) is called. That call leaves the final string on the top of the stack.

During its normal operation, a string buffer uses a variable number of stack slots. So, while using a buffer, the developer shall not assume that the level of the top of the stack is known. The stack between successive calls may be used to buffer operations as long as that use is balanced; i.e., when a buffer operation is called, the stack is at the same level it was immediately after the previous buffer operation. (The only exception to this rule is luaL_addvalue.) After calling luaL_pushresult, the stack is back to its level when the buffer was initialized, plus the final string on its top.

---

luaL_buffinit

```
void luaL_buffinit (lua_State *L, luaL_Buffer *B);
```

initializes a buffer B.

This function does not allocate any space; the buffer must be declared as a variable (see luaL_Buffer).

---

luaL_callmeta

```
int luaL_callmeta (lua_State *L, int obj, const char *e);
```

calls a metamethod.

If the object at index obj has a metatable and that metatable has a field e, this function calls that field and passes the object as its only argument. In that case, this function returns 1 and pushes the value returned by the call on to the stack. If there is no metatable or no metamethod, this function returns 0 (without pushing any value on the stack).

---

luaL_checkany

```
void luaL_checkany (lua_State *L, int narg);
```

checks whether the function has an argument of any type (including nil) at position narg.

---

luaL_checkint

```
int luaL_checkint (lua_State *L, int narg);
```

checks whether the function argument narg is a number and returns that number cast to an int.

---

luaL_checkinteger

```
lua_Integer luaL_checkinteger (lua_State *L, int narg);
```

checks whether the function argument narg is a number and returns that number cast to a lua_Integer.

---

luaL_checklong

```
long luaL_checklong (lua_State *L, int narg);
```

checks whether the function argument narg is a number and returns that number cast to a long.

---

luaL_checklstring

```
const char *luaL_checklstring (lua_State *L, int narg, size_t *l);
```

checks whether the function argument narg is a string and returns that string; if l is not NULL fills *l with the string's length.

luaL_checknumber

```
lua_Number luaL_checknumber (lua_State *L, int narg);
```

checks whether the function argument narg is a number and returns that number.

luaL_checkoption

```
int luaL_checkoption (lua_State *L, int narg, const char *def, const
char *const lst[]);
```

checks whether the function argument narg is a string and searches for that string in the array lst (which must be NULL-terminated). If def is not NULL, uses def as a default value when the function has no argument narg or if that argument is nil.

This function returns the index in the array where the string was found, or raises an error if the argument is not a string or if the string cannot be found.

This is a useful function for mapping strings to C enums. The usual convention in Lua libraries is to use strings instead of numbers to select options.

luaL_checkstack

```
void luaL_checkstack (lua_State *L, int sz, const char *msg);
```

grows the stack size to top + sz elements, raising an error if the stack cannot grow to that size. msg is an additional text to go into the error message.

luaL_checkstring

```
const char *luaL_checkstring (lua_State *L, int narg);
```

checks whether the function argument narg is a string and returns that string.

luaL_checktype

```
void luaL_checktype (lua_State *L, int narg, int t);
```

checks whether the function argument narg has type t.

luaL_checkudata

```
void *luaL_checkudata (lua_State *L, int narg, const char *tname);
```

checks whether the function argument narg is a userdata of the type tname (see luaL_newmetatable).

luaL_error

```
int luaL_error (lua_State *L, const char *fmt, ...);
```

raises an error.

The error message format is given by fmt plus any extra arguments, following the same rules of lua_pushfstring. It also adds the file name and the line number where the error occurred, if that information is available, at the beginning of the message.

This function never returns, but it is an idiom to use it as return luaL_error ... in C functions.

luaL_getmetafield

```
int luaL_getmetafield (lua_State *L, int obj, const char *e);
```

pushes the field e from the metatable of the object at index obj on to the stack. If the object does not have a metatable, or if the metatable does not have that field, returns 0 and pushes nothing.

luaL_getmetatable

```
void luaL_getmetatable (lua_State *L, const char *tname);
```

pushes the metatable associated with name tname in the registry (see luaL_newmetatable) on to the stack.

luaL_gsub

```
const char *luaL_gsub (lua_State *L, const char *s, const char *p,
const char *r);
```

creates a copy of string s by replacing any occurrence of the string p with the string r, pushes the resulting string on to the stack and returns it.

luaL_loadbuffer

```
int luaL_loadbuffer (lua_State *L, const char *buff, size_t sz, const
char *name);
```

loads a buffer as a Lua chunk.

This function uses lua_load to load the chunk in the buffer pointed by buff with size sz.

This function returns the same results as lua_load. name Is the chunk name, used for debug information and error messages.

luaL_loadfile

```
int luaL_loadfile (lua_State *L, const char *filename);
```

loads a file as a Lua chunk.

This function uses lua_load to load the chunk in the file named filename. If filename is NULL, then it loads from the standard input. The first line in the file is ignored if it starts with a #.

This function returns the same results as lua_load, but it has an extra error code LUA_ERRFILE if it cannot open or read the file.

luaL_loadstring

```
int luaL_loadstring (lua_State *L, const char *s);
```

loads a string as a Lua chunk.

This function uses lua_load to load the chunk in the zero-terminated string s.

This function returns the same results as lua_load.

luaL_newmetatable

```
int luaL_newmetatable (lua_State *L, const char *tname);
```

returns 0 if the registry already has the key tname; otherwise, creates a new table to be used as a metatable for userdata, adds it to the registry with key tname and returns 1.

In both cases, this function pushes the final value associated with tname in the registry on to the stack.

luaL_newstate

```
lua_State *luaL_newstate (void);
```

creates a new Lua state, calling lua_newstate with an allocation function based on the standard C realloc function and setting a panic function (see lua_atpanic) that prints an error message to the standard error output in the case of fatal errors.

This function returns the new state or NULL, if there is a memory allocation error.

---

luaL_openlibs

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

opens all standard Lua libraries into the given state.

---

luaL_optint

```
int luaL_optint (lua_State *L, int narg, int d);
```

if the function argument narg is a number, returns that number cast to an int. If that argument is absent or is nil, this function returns d; otherwise, it raises an error.

---

luaL_optinteger

```
lua_Integer luaL_optinteger (lua_State *L, int narg, lua_Integer d);
```

if the function argument narg is a number, returns that number cast to a lua_Integer. If that argument is absent or is nil, this function returns d; otherwise, it raises an error.

---

luaL_optlong

```
long luaL_optlong (lua_State *L, int narg, long d);
```

if the function argument narg is a number, returns that number cast to a long. If that argument is absent or is nil, this function returns d; otherwise, it raises an error.

---

luaL_optlstring

```
const char *luaL_optlstring (lua_State *L, int narg, const char *d,
size_t *l);
```

if the function argument narg is a string, returns that string. If that argument is absent or is nil, this function returns d; otherwise, it raises an error.

If l is not NULL, fills the position *l with the result length.

---

luaL_optnumber

```
lua_Number luaL_optnumber (lua_State *L, int narg, lua_Number d);
```

if the function argument narg is a number, returns that number. If that argument is absent or is nil, returns d. Otherwise, raises an error.

---

luaL_optstring

```
const char *luaL_optstring (lua_State *L, int narg, const char *d);
```

If the function argument narg is a string, returns that string. If that argument is absent or is nil, this function returns d; otherwise, it raises an error.

---

luaL_prepbuffer

```
char *luaL_prepbuffer (luaL_Buffer *B);
```

returns an address to a space of size LUAL_BUFFERSIZE where the developer may copy a string to be added to buffer B (see luaL_Buffer).

After copying the string into that space, luaL_addsize with the size of the string must be called to actually add it to the buffer.

luaL_pushresult

```
void luaL_pushresult (luaL_Buffer *B);
```

finishes the use of buffer B leaving the final string on the top of the stack.

luaL_ref

```
int luaL_ref (lua_State *L, int t);
```

creates and returns a reference, in the table at index t, for the object at the top of the stack (and pops the object).

A reference is a unique integer key. As long as the developer does not manually add integer keys into table t, luaL_ref ensures the uniqueness of the key it returns. An object referred to by reference r may be retrieved by calling lua_rawgeti(L, t, r). Function luaL_unref frees a reference and its associated object.

If the object at the top of the stack is nil, luaL_ref returns the constant LUA_REFNIL. The constant LUA_NOREF is guaranteed to be different from any reference returned by luaL_ref.

luaL_Reg

```
typedef struct luaL_Reg {
  const char *name;
  lua_CFunction func;
} luaL_Reg;
```

Is a type for arrays of functions to be registered by luaL_register. name Is the function name and func is a pointer to the function. Any array of luaL_Reg must end with a sentinel entry in which both name and func are NULL.

luaL_register

```
void luaL_register (lua_State *L, const char *libname, const luaL_Reg
*l);
```

opens a library.

When called with libname equal to NULL, simply registers all functions in the list l (see luaL_Reg) into the table on the top of the stack.

When called with a non-null libname, creates a new table t, sets it as the value of the global variable libname, sets it as the value of package.loaded[libname], and registers on it all functions in the list l. If there is a table in package.loaded[libname] or in variable libname, reuses that table instead of creating a new one.

In any case, the function leaves the table on the top of the stack.

luaL_typename

```
const char *luaL_typename (lua_State *L, int idx);
```

returns the name of the type of the value at index idx.

luaL_typerror

```
int luaL_typerror (lua_State *L, int narg, const char *tname);
```

generates an error with a message like

<location>: bad argument <narg> to <function> (<tname> expected, got <realt>)

where <location> is produced by luaL_where, <function> is the name of the current function and <realt> is the type name of the actual argument.

luaL_unref

```
void luaL_unref (lua_State *L, int t, int ref);
```

releases reference ref from the table at index t (see luaL_ref). The entry is removed from the table, so that the referred object may be collected. The reference ref is also freed to be used again.

If ref is LUA_NOREF or LUA_REFNIL, luaL_unref does nothing.

luaL_where

```
void luaL_where (lua_State *L, int lvl);
```

pushes a string identifying the current position of the control at level lvl in the call stack on to the stack. Typically this string has the format <chunkname>:<currentline>:. Level 0 is the running function, level 1 is the function that called the running function, etc.

This function is used to build a prefix for error messages.

A.4 Standard libraries

A.4.1 Overview

The standard Lua 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 requirements that 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;

– package library;

– string manipulation;

– table manipulation;

– mathematical functions (sin, log, etc.);

– input and output;

– operating system facilities;

– debug facilities.

Except for the basic and package libraries, 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 call luaL_openlibs, which opens all standard libraries. Alternatively, it may open them individually by calling luaopen_base (for the basic library), luaopen_package (for the package 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 and should not be called directly: they must be called like any other Lua C function, e.g., by using lua_call.

A.4.2 Basic functions

The basic library provides some core functions to Lua. If this library is not included in the application, the developer should check carefully whether implementations need to be provided for some of its facilities.

assert (v [, message])

issues an error when the value of its argument v is false (i.e., nil or false); otherwise, returns all its arguments. message Is an error message; when absent, it defaults to "assertion failed!"

collectgarbage (opt [, arg])

is a generic interface to the garbage collector.

It performs different functions according to its first argument, opt, as follows.

– "stop" --- Stops the garbage collector.

– "restart" --- Restarts the garbage collector.

– "collect" --- Performs a full garbage-collection cycle.

– "count" --- Returns the total memory in use by Lua (in Kbytes).

– "step" --- Performs a garbage-collection step. The step "size" is controlled by arg (larger values mean more steps) in a non-specified way. If the developer wants to control the step size, the value of arg must be tuned experimentally. This function returns true if that step finished a collection cycle.

– "steppause" --- Sets arg/100 as the new value for the pause of the collector (see clause A.1.11).

– "setstepmul" --- Sets arg/100 as the new value for the step multiplier of the collector (see clause A.1.11).

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). This function returns all values returned by the chunk. In the case of errors, dofile propagates the error to its caller (i.e., dofile 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.

Usually, error adds some information about the error position at the beginning of the message. The level argument specifies how to get the error position. 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. Passing a level 0 avoids the addition of error position information to the message.

_G

is a global variable (not a function) that holds the global environment (i.e., _G._G = _G).

Lua itself does not use this variable; changing its value does not affect any environment, nor vice-versa. (Use setfenv to change environments.)

getfenv (f)

returns the current environment in use by the function.

f May be a Lua function or a number that 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.

---

getmetatable (object)

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

---

ipairs (t)

returns three values: 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.

See next for the caveats of modifying the table during its traversal.

---

load (func [, chunkname])

loads a chunk using function func to get its pieces.

Each call to func must return a string that concatenates with previous results. A return of nil (or no value) signals the end of the chunk.

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.

chunkname is used as the chunk name for error messages and debug information.

---

loadfile ([filename])

is similar to load, but gets the chunk from file filename or from the standard input if no file name is given.

---

loadstring (string [, chunkname])

is similar to load, but gets the chunk from the given string.

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 its associated value. When called with nil as its second argument, next returns an initial index 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. In particular, next(t) may be used to check whether a table is empty.

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 behaviour of next is undefined if, during the traversal, any value is assigned to a non-existent field in the table. However, existing fields may be modified. In particular, existing fields may be cleared.

---

pairs (t)

returns three values: the next function, the table t, and nil, so that the construction

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

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

See next for the caveats of modifying the table during its traversal.

---

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 a case, pcall also returns all results from the call after this first result. In the 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.

print Is not intended for formatted output, but only as a quick way to show a value, typically for debugging. For formatted output, use string.format.

---

rawequal (v1, v2)

checks whether v1 is equal to v2, without invoking any metamethod and returns a boolean.

---

rawget (table, index)

gets the real value of table[index], without invoking any metamethod.

table Must be a table and index 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 any value different from nil, and value any Lua value.

---

select (index, ...)

returns all arguments after argument number index, if index is a number.

Otherwise, index must be the string "#" and select returns the total number of extra arguments it received.

---

setfenv (f, table)

sets the environment to be used by the given function.

f May be a Lua function or a number that specifies the function at that stack level: Level 1 is the function calling setfenv. setfenv Returns the given function.

As a special case, when f is 0, setfenv changes the environment of the running thread. In this case, setfenv returns no values.

---

setmetatable (table, metatable)

sets the metatable for the given table. (The metatable of other types shall not be changed from Lua, only from C.)

If metatable is nil, removes the metatable of the given table. If the original metatable has a "__metatable" field, raises an error.

This function returns a table.

---

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 clause A.1.2). 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 string.format.

If the metatable of e has a "__tostring" field, then 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 [, i [, j]])

returns the elements from the given table.

This function is equivalent to

```
return list[i], list[i+1], ..., list[j]
```

except that the above code may be written only for a fixed number of elements. By default, i is 1 and j is the length of the list, as defined by the length operator (see clause A.1.6.6).

_VERSION

is a global variable (not a function) that holds a string containing the current interpreter version.

The current content of this variable is "Lua 5.1".

---

xpcall (f, err)

is a function is similar to pcall, except that a new error handler may be set.

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 this case, xpcall also returns all results from the call, after this first result. In the case of any error, xpcall returns false plus the result from err.

A.4.3 Coroutine manipulation

The operations related to coroutines comprise a sub-library of the basic library and come inside the table coroutine. See clause A.1.12 for a general description of coroutines.

---

coroutine.create (f)

creates a new coroutine, with body f.

f Must be a Lua function. This function returns this new coroutine, an object with type "thread".

---

coroutine.resume (co [, val1, ..., valn])

starts or continues the execution of coroutine co.

The first time a coroutine is resumed, it starts running its body. The values val1, ..., valn are passed as the arguments to the body function. If the coroutine has yielded, resume restarts it; the values val1, ..., valn are passed 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.running ()

returns the running coroutine or nil when called by the main thread.

---

coroutine.status (co)

returns the status of coroutine co, as a string: "running", if the coroutine is running (i.e., it called status); "suspended", if the coroutine is suspended in a call to yield or if it has not started running yet; "normal" if the coroutine is active, but not running (i.e., it has resumed another coroutine); 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. This 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 the case of error, propagates the error.

---

coroutine.yield ([val1, ..., valn])

suspends the execution of the calling coroutine.

The coroutine may not run a C function, a metamethod or an iterator. Any arguments to yield are passed as extra results to resume.

A.4.4 Modules

The package library provides basic facilities for loading and building modules in Lua. It exports two of its functions directly in the global environment: require and module. Everything else is exported in a table package.

---

module (name [, ...])

creates a module.

If there is a table in package.loaded[name], that table is the module. Otherwise, if there is a global table t with the given name, that table is the module. Otherwise creates a new table t and sets it as the value of the global name and the value of package.loaded[name]. This function also initializes t._NAME with the given name, t._M with the module (t itself), and t._PACKAGE with the package name (the full module name minus last component; see the following). Finally, module sets t as the new environment of the current function and the new value of package.loaded[name], so that require returns t.

If name is a compound name (i.e., one with components separated by dots), module creates (or reuses, if they already exist) tables for each component. For instance, if name is a.b.c, then module stores the module table in field c of field b of global a.

This function may receive optional options after the module name, where each option is a function to be applied over the module.

---

require (modname)

loads the given module.

The function starts by looking into the table package.loaded to determine whether modname is already loaded. If it is, then require returns the value stored at package.loaded[modname]. Otherwise, it tries to find a loader for that module.

To find a loader, first require queries package.preload[modname]. If it has a value, that value (which should be a function) is the loader. Otherwise require searches for a Lua loader using the path stored in package.path. If that also fails, it searches for a C loader using the path stored in package.cpath. If that also fails, it tries an all-in-one loader (see the following).

When loading a C library, require first uses a dynamic link facility to link the application with the library. Then it tries to find a C function inside that library to be used as the loader. The name of that C function is the string "luaopen_" concatenated with a copy of the module name where each dot is replaced by an underscore. Moreover, if the module name has a hyphen, its prefix up to (and including) the first hyphen is removed. For instance, if the module name is a.v1-b.c, the function name will be luaopen_b_c.

If require finds neither a Lua library nor a C library for a module, it calls the all-in-one loader. That loader searches the C path for a library for the root name of the given module. For instance, when requiring a.b.c, it will search for a C library for a. If found, it looks into it for an open function for the submodule; in our example, that would be luaopen_a_b_c. With that facility, a package may pack several C submodules into one single library, with each submodule keeping its original open function.

Once a loader is found, require calls the loader with a single argument, modname. If the loader returns any value, require assigns it to package.loaded[modname]. If the loader returns no value and

has not assigned any value to package.loaded[modname], then require assigns true to that entry. In any case, require returns the final value of package.loaded[modname].

If there is any error loading or running the module, or if it cannot find any loader for that module, then require signals an error.

---

package.cpath

is the path used by require to search for a C loader.

Lua initializes the C path package.cpath in the same way it initializes the Lua path package.path, using the environment variable LUA_CPATH (plus another default path defined in luaconf.h [b-luaconf.h]).

---

package.loaded

is a table used by require to control which modules are already loaded. When a module modname is required and package.loaded[modname] is not false, require simply returns the value stored there.

---

package.loadlib (libname, funcname)

dynamically links the host program with the C library libname.

Inside this library, looks for a function funcname and returns this function as a C function. [So, funcname must follow the protocol (see lua_CFunction)].

This is a low-level function. It completely bypasses the package and module system. Unlike require, it does not perform any path searching and does not automatically add extensions. libname Must be the complete file name of the C library, including if necessary a path and extension. funcname Must be the exact name exported by the C library (which may depend on the C compiler and linker used).

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

---

package.path

is the path used by require to search for a Lua loader.

At start-up, Lua initializes this variable with the value of the environment variable LUA_PATH or with a default path defined in luaconf.h [b-luaconf.h], if the environment variable is not defined. Any ";;" in the value of the environment variable is replaced by the default path.

A path is a sequence of templates separated by semicolons. For each template, require will change each interrogation mark in the template by filename, which is modname with each dot replaced by a "directory separator" (such as "/" in Unix); then it will try to load the resulting file name. So, for instance, if the Lua path is

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

the search for a Lua loader for module foo will try to load the files ./foo.lua, ./foo.lc, and /usr/local/foo/init.lua, in that order.

---

package.preload

is a table to store loaders for specific modules (see require).

package.seeall (module)

sets a metatable for module with its __index field referring to the global environment, so that this module inherits values from the global environment. To be used as an option to function module.

A.4.5 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. It also sets a metatable for strings where the __index field points to the metatable itself. Therefore, the string functions in object-oriented style may be used. For instance, string.byte(s, i) may be written as s:byte(i).

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

returns the internal numerical codes of the characters s[i], s[i+1], ..., s[j].

The default value for i is 1; the default value for j is i.

Note that numerical codes are not necessarily portable across platforms.

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

receives 0 or more integers.

This function returns a string with length equal to the number of arguments, in which each character has the internal numerical code equal to its corresponding argument.

Note that numerical codes are not necessarily portable across platforms.

string.dump (function)

returns a string containing 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 a match, then find returns the indices of s where this occurrence starts and ends; otherwise, it returns nil. A third, optional numerical argument init specifies where to start the search; its default value is 1 and may be negative. 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 as well.

If the pattern has captures, then in a successful match the captured values are also returned, after the two indices.

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 that 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, embedded zeros 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.

This function does not accept string values containing embedded zeros.

---

string.gmatch (s, pattern)

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

If pattern 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.gmatch(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.gmatch(s, "(%w+)=(%w+)") do
  t[k] = v
end
```

---

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

returns a copy of s in which all occurrences of the pattern have been replaced by a replacement string specified by repl, which may be a string, a table, or a function.

gsub Also returns, as its second value, the total number of substitutions made.

If repl is a string, then its value is used for replacement. The character % works as an escape character. Any sequence in repl of the form %n, with n between 1 and 9, stands for the value of the nth captured substring (see the following). The sequence %0 stands for the whole match. The sequence %% stands for a single %.

If repl is a table, then the table is queried for every match, using the first capture as the key; if the pattern specifies no captures, then the whole match is used as the key.

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 the table query or by the function call is a string or a number, then it is used as the replacement string; otherwise, if it is false or nil, then there is no replacement (i.e., the original match is kept in the 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 pattern 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+", "%0 %0", 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.getenv)
--> 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="5.1"}
x = string.gsub("$name%-$version.tar.gz", "%$(%w+)", t)
--> x="lua-5.1.tar.gz"
```

---

string.len (s)

receives a string and returns its length.

The empty string "" has length 0. Embedded zeros are counted, so "a\000bc\000" 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 an uppercase letter is depends on the current locale.

---

string.match (s, pattern [, init])

looks for the first match of pattern in the string s.

If it finds one, then match returns the captures from the pattern; otherwise it returns nil. If pattern specifies no captures, then the whole match is returned. A third, optional numerical argument init specifies where to start the search; its default value is 1 and may be negative.

---

string.rep (s, n)

returns a string that is the concatenation of n copies of the string s.

---

string.reverse (s)

returns a string that is the string s reversed.

---

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 a lowercase letter is depends on the current locale.

Patterns

is a character class 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) may 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 in the foregoing 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 in the foregoing.

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] cannot be equivalent to %l.

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 nth captured string (see the following);

– %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 the string is read 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 the pattern "()aa()" is applied on the string "flaaap", there will be two captures: 3 and 5.

A pattern shall not contain embedded zeros. Use %z instead.

A.4.6 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, reference to the "length" of a table means the result of the length operator.

---

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 length of the table. If i is greater than j, returns the empty string.

---

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 length of the table (see clause A.1.6.6), so that a call table.insert(t,x) inserts x at the end of table t.

---

table.maxn (table)

returns the largest positive numerical index of the given table, or zero if the table has no positive numerical indices. (To do its job this function does a linear traversal of the whole table.)

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 length of the table, so that a call table.remove(t) removes the last element of table t.

table.sort (table [, comp])

sorts table elements in a given order, in-place, from table[1] to table[n], where n is the length of the table. 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; i.e., elements considered equal by the given order may have their relative positions changed by the sort.

A.4.7 Mathematical functions

This library is an interface to the standard C math library. It provides all its functions inside the table math. The library provides the functions listed in Table A.3.

Table A.3

math.abs	math.acos	math.asin	math.atan	math.atan2
math.ceil	math.cos	math.cosh	math.deg	math.exp
math.floor	math.fmod	math.frexp	math.ldexp	math.log
math.log10	math.max	math.min	math.modf	math.pow
math.rad	math.random	math.randomseed	math.sin	math.sinh
math.sqrt	math.tan	math.tanh		

In addition to the functions in Table A.3, there are a variable math.pi and a variable math.huge, with the value HUGE_VAL. Most of those functions are only interfaces to the corresponding functions in the C library. All trigonometric functions work in radians. 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 may be used with 1, 2 or more arguments.

The function math.modf corresponds to the modf C function. It returns two values: The integral part and the fractional part of its argument. The function math.frexp also returns two values: The normalized fraction and the exponent of its argument.

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.

A.4.8 Input and output facilities

The input/output (I/O) library provides two different styles for file manipulation. The first uses implicit file descriptors; i.e., there are operations to set a default input file and a default output file, and all I/O 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 of 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.

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, 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 the 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(); i.e., it iterates over the lines of the default input file. In that case, it does not close the file when the loop ends.

---

io.open (filename [, mode])

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 may 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])

is similar to io.input, but operates over the default output file.

---

io.popen ([prog [, mode]])

starts program prog in a separated process and returns a file handle that may be used to read data from that program (if mode is "r", the default) or to write data to that program (if mode is "w").

This function is system dependent and is not available on all platforms.

---

io.read (format1, ...)

equivalent to io.input():read.

---

io.tmpfile ()

returns a handle for a temporary file.

This file is opened in update mode and it is automatically removed when the program ends.

---

io.type (obj)

checks whether obj is a valid file handle.

This function 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.

Note that files are automatically closed when their handles are garbage collected, but that takes an unpredictable amount of time to happen.

---

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 the following).

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 the 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:setvbuf (mode [, size])

sets the buffering mode for an output file.

There are three available modes:

– "no" --- no buffering; the result of any output operation appears immediately;

– "full" --- full buffering; output operation is performed only when the buffer is full [or when the file is explicitly flushed (see clause 9.8)];

– "line" --- line buffering; output is buffered until a newline is output or there is any input from some special files (such as a terminal device).

For the last two cases, sizes specifies the size of the buffer, in bytes. The default is an appropriate size.

file:write (value1, ...)

writes the value of each of its arguments to the file.

The arguments must be strings or numbers. To write other values, use tostring or string.format before write.

A.4.9 Operating system facilities

This library is implemented through table os.

---

os.clock ()

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

---

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 [i.e., 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])

is equivalent to the C function system.

This function passes command to be executed by an operating system shell. It returns a status code, which is system-dependent. If command is absent, then it returns nonzero if a shell is available and zero otherwise.

---

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 or directory with the given name.

Directories must be empty to be removed. If this function fails, it returns nil, plus a string describing the error.

os.rename (oldname, newname)

renames file or directory 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 honoured.

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 the 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 may be used only as an argument to date and difftime.

os.tmpname ()

returns a string with a file name that may be used for a temporary file.

The file must be explicitly opened before its use and explicitly removed when no longer needed.

A.4.10 The debug library

This library provides the functionality of the debug interface to Lua programs. Care should be exercised when using this library. The functions provided here should be used exclusively for debugging and similar tasks, such as profiling. Developers should resist the temptation to use them as a usual programming tool: They can be very slow. Moreover, several functions violate some assumptions about Lua code (e.g., that variables local to a function cannot be accessed from outside or that userdata metatables cannot be changed by Lua code) and therefore can compromise otherwise secure code.

All functions in this library are provided inside the 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 may 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 within any function, and so have no direct access to local variables.

debug.getfenv (o)

returns the environment of object o.

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])

returns a table with information about a function.

The function may be given directly, or a number may be given 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 a 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)

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. (debug.getinfo May be called to check whether the level is valid.)

Variable names starting with `(´ (open parenthesis) represent internal variables (loop control variables, temporaries and C function locals).

debug.getmetatable (object)

returns the metatable of the given object or nil if it does not have a metatable.

debug.getregistry ()

returns the registry table (see clause A.2.6).

debug.getupvalue (func, up)

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.setfenv (object, table)

sets the environment of the given object to the given table.

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, debug.sethook turns off the hook.

When the hook is called, its first parameter is a string describing the event that has triggered its call: "call", "return" (or "tail return"), "line" and "count". For line events, the hook also gets the new line number as its second parameter. Inside a hook, getinfo may be called 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.setlocal (level, local, value)

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. (getinfo May be called to check whether the level is valid.) Otherwise, it returns the name of the local variable.

---

debug.setmetatable (object, table)

sets the metatable for the given object to the given table (which may be nil).

---

debug.setupvalue (func, up, value)

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. Otherwise, it returns the name of the upvalue.

---

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.

A.5 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, including the debug library. Its usage is:

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

The options are:

– -e stat executes string stat;

– -l mod "requires" mod;

– -i enters interactive mode after running script;

– -v prints version information;

– -- stops handling options;

– executes stdin as a file and stops handling options.

After handling its options, lua runs the given script, passing to it the given args as string arguments. When called without arguments, lua behaves as lua -v -i when the standard input (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 the value of a (which is `1´), and finally run the file script.lua with no arguments. (Here $ is the shell prompt. The prompt may be different in different implementations.)

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 at index 0, the first argument after the script name goes to index 1 and so on. Any arguments before the script name (i.e., the interpreter name plus the options) go to negative indices. For instance, in the call

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

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

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

and finally runs the file b.lua. The script is called with arg[1], arg[2], ... as arguments; it may also access those arguments with the vararg expression `...´.

If an incomplete statement is written in interactive mode, the interpreter waits for its completion by issuing a different prompt.

If the global variable _PROMPT contains a string, then its value is used as the prompt. Similarly, if the global variable _PROMPT2 contains a string, its value is used as the secondary prompt (issued during incomplete statements). Therefore, both prompts may be changed directly on the command line. For instance,

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

(the outer pair of quotes is for the shell, the inner pair 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 just end silently right after the assignment to _PROMPT.

To allow the use of Lua as a script interpreter in Unix systems, the stand-alone interpreter skips the first line of a chunk if it starts with #. Therefore, Lua scripts may 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 can be different in the user's machine. If lua is in the PATH, then

```
#!/usr/bin/env lua
```

is a more portable solution.)

A.6 Incompatibilities with the version 5.0

NOTE – This clause lists the incompatibilities that can be found when moving a program from Lua 5.0 to Lua 5.1. Most incompatibilities in compiling Lua can be avoided with appropriate options (see file luaconf.h [b-luaconf.h]). However, all those compatibility options will be removed in the next version of Lua.

A.6.1 Changes in the language

These are the changes in the language introduced by Lua 5.1.

– The vararg system has changed from the pseudo-argument arg with a table with the extra arguments to the vararg expression. (Option LUA_COMPAT_VARARG in luaconf.h [b-luaconf.h].)

– There has been a subtle change in the scope of the implicit variables of the for statement and for the repeat statement.

– The long string/long comment syntax ([[...]]) does not allow nesting. The new syntax ([=[...]=]) may be used in those cases. (Option LUA_COMPAT_LSTR in luaconf.h [b-luaconf.h].)

A.6.2 Changes in the libraries

These are the changes in the libraries introduced by Lua 5.1.

– Function string.gfind was renamed string.gmatch (Option LUA_COMPAT_GFIND).

– When string.gsub is called with a function as its third argument, whenever that function returns nil or false the replacement string is the whole match, instead of the empty string.

– Function table.setn was deprecated. Function table.getn corresponds to the new length operator (#); use the operator instead of the function (Option LUA_COMPAT_GETN).

– Function loadlib was renamed package.loadlib (Option LUA_COMPAT_LOADLIB).

– Function math.mod was renamed math.fmod (Option LUA_COMPAT_MOD).

– There were substantial changes in function required due to the new module system. However, the new behaviour is mostly compatible with the old, but require gets the path from package.path instead of from LUA_PATH.

– Function collectgarbage has different arguments. Function gcinfo is deprecated; use collectgarbage("count") instead.

A.6.3 Changes in the API

These are the changes in the C API introduced by Lua 5.1.

– The luaopen_* functions (to open libraries) shall not be called directly, like a regular C function. They must be called through Lua, like a Lua function.

– Function lua_open was replaced by lua_newstate to allow the user to set a memory allocation function. luaL_newstate From the standard library may be used to create a state with a standard allocation function (based on realloc).

– Functions luaL_getn and luaL_setn (from the auxiliary library) are deprecated. Use lua_objlen instead of luaL_getn and nothing instead of luaL_setn.

– Function luaL_openlib was replaced by luaL_register.

A.7 The complete syntax of Lua

Here is the complete syntax of Lua in extended BNF. It does not describe operator priorities or some syntactical restrictions, such as return, and break statements may only appear as the last statement of a block.

```
chunk ::= {stat [`;´]} [laststat[`;´]]
block ::= chunk
stat ::=  varlist1 `=´ explist1  |
    functioncall  |
```

```
    do block end  |
    while exp do block end  |
    repeat block until exp  |
    if exp then block {elseif exp then block}[else block] end |
    for Name `=´ exp `,´ exp [`,´ exp] do block end  |
    for namelist in explist1 do block end  |
    function funcname funcbody  |
    local function Name funcbody  |
    local namelist [`=´ explist1]
laststat ::= return [explist1]  |  break
funcname ::= Name {`.´ Name} [`:´ Name]
varlist1 ::= var {`,´ var}
var ::=  Name  |  prefixexp `[´ exp `]´  |  prefixexp `.´ Name
namelist ::= Name {`,´ Name}
explist1 ::= {exp `,´} exp
exp ::= nil  |  false  |  true  |  Number  |  String  |  `...´  |
     function  |  prefixexp  |  tableconstructor  |
     exp binop |  exp  |  unop exp
prefixexp ::= var  |  functioncall  |  `(´ exp `)´
functioncall ::=  prefixexp args  |  prefixexp `:´ Name args
args ::=  `(´ [explist1] `)´  |  tableconstructor  |  String
function ::= function funcbody
funcbody ::= `(´ [parlist1] `)´ block end
parlist1 ::= namelist [`,´ `...´]  |  `...´
tableconstructor ::= `{´ [fieldlist] `}´
fieldlist ::= field {fieldsep field} [fieldsep]
field ::= `[´ exp `]´ `=´ exp  |  Name `=´ exp  |  exp
fieldsep ::= `,´  |  `;´
binop ::= `+´  |  `-´  |  `*´  |  `/´  |  `^´  |  `%´  |  `..´  |
       `<´  |  `<=´  |  `>´  |  `>=´  |  `==´  |  `~=´  |
       and  |  or
unop ::= `-´  |  not  |  `#´
```

Appendix I

Lua scripting examples

I.1 Example using canvas and event modules

This scripting example is a "maximum healthy weight calculator" that takes into account the body mass index.

```
--------------------------------------------------------------------------------
-- DRAWS: this code excerpt may be reused in other applications
--------------------------------------------------------------------------------

local DX, DY = canvas:attrSize()         -- screen dimensions
local DRAWS = {}                         -- drawing list

function redraw ()                       -- list redrawing function
    for _, draw in ipairs(DRAWS) do
        draw:draw()
    end
    canvas:flush()
end

function drawRect (self)                 -- rectangle drawing function
    canvas:attrColor(self.color)
    canvas:drawRect('fill', self.x, self.y, self.dx, self.dy)
end

function drawText (self)                 -- text drawing function
    canvas:attrColor(self.color)
    canvas:attrFont(self.face, self.dy)
    canvas:drawText(self.x, self.y, self.text)
end

function drawImage (self)                -- image drawing function
    canvas:compose(self.x, self.y, self.cvs)
end

--------------------------------------------------------------------------------
-- IDEAL WEIGHT CALCULATOR
--------------------------------------------------------------------------------

local PADDING = 15
local APP
local HEIGHT = ''
local INPUT

-- Black background
DRAWS[#DRAWS+1] = {
    color = 'black',
    draw  = drawRect,
    x = 0, dx = DX,
    y = 0, dy = DY,
}

-- background image
--[[
DRAWS[#DRAWS+1] = {
    cvs  = canvas:new('3dbox.png'),
    draw = drawImage,
    x = 0, y = 0,
}
```

```
]]

local END = false

function hdlr_enter (evt)
    if END then return end
    if evt.class ~= 'key'   then return end
    if evt.type  ~= 'press' then return end
    if evt.key == 'ENTER' then
        assert(coroutine.resume(APP))
    end
end

function hdlr_input (evt)
    if END then return end
    if evt.class ~= 'key'   then return end
    if evt.type  ~= 'press' then return end
    local key = evt.key
    if tonumber(key) then
        HEIGHT = HEIGHT .. key
    elseif key == 'CURSOR_LEFT' then
        HEIGHT = string.sub(HEIGHT, 1, -2)
    end
    INPUT.text = HEIGHT
    redraw()
end

APP = coroutine.create(
function()
    local message = {
        text  = 'Enter your height in centimetres (Ex: 165):',
        face  = 'vera',
        color = 'white',
        draw  = drawText,
        x = PADDING,
        y = PADDING, dy = 20,
    }
    DRAWS[#DRAWS+1] = message

    INPUT = {
        text  = HEIGHT,
        face  = 'vera',
        color = 'white',
        draw  = drawText,
        x  = message.x,
        y  = message.y+message.dy+PADDING,
        dy = 20
    }
    DRAWS[#DRAWS+1] = INPUT
    redraw()

    event.register(hdlr_enter)
    event.register(hdlr_input)
    coroutine.yield()      -- wait for ENTER

    DRAWS[#DRAWS] = nil
    DRAWS[#DRAWS] = nil
    redraw()

    local answer
    if tonumber(HEIGHT) then
        answer = (HEIGHT/100)^2 * 25
        answer = "Your maximum weight is "..string.format("%3.2f", answer)
    else
```

```
        answer = "Invalid height!"
    end

    local message = {
        text  = answer,
        face  = 'vera',
        color = 'white',
        draw  = drawText,
        x = PADDING,
        y = PADDING, dy = 20,
    }
    DRAWS[#DRAWS+1] = message
    redraw()

    coroutine.yield()      -- wait for ENTER
    -- may notify host language that the calculation reached an end
    END = true
end)

-- may be started as an action from host language, anyway:
assert(coroutine.resume(APP))
```

Bibliography

[b-ITU-T J.200] Recommendation ITU-T J.200 (2010), *Worldwide common core – Application environment for digital interactive television services*.

[b-Lua 5.1] Ierusalimschy, R., de Figueiredo, L.H., Celes, W. (2006). *Lua 5.1 reference manual*. by Rio de Janeiro: Lua.org, PUC-Rio. Available (viewed 2019-02-21) at: https://www.lua.org/manual/5.1/

[b-luac] b-luac. Available (viewed 2019-02-27) at: https://www.lua.org/manual/5.1/luac.html

[b-luaconf.h] luaconf.h. Available (viewed 2019-02-27) at: https://www.lua.org/source/5.1/luaconf.h.html

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	Tariff and accounting principles and international telecommunication/ICT economic and policy issues
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	**Audiovisual and multimedia systems**
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Environment and ICTs, climate change, e-waste, energy efficiency; construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling, and associated measurements and tests
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities
Series Z	Languages and general software aspects for telecommunication systems

Printed in Switzerland
Geneva, 2019