Gates

Paul Isambert
zappathustra@free.fr
Version 0.2
May 2012

Gates : a presentation

This part of the documentation is a general introduction to gates, illustrating many of their properties in the abstract. Then comes *a user manual for TEX*, *a complete reference for TEX*, *a user manual for Lua* and *a complete reference for Lua*. The manuals explain gates in action, while the references are alphabetical lists of all actions with their syntax.

This documentation doesn't contain very complex examples of gates. To see gates in full regalia, you can have a look at PiTEX (the set of files used to typeset this documentation) for (mostly) TEX gates, and the Interpreter package, version 1.1, for Lua gates (version 1.0, wasn't written with Gates). Speaking of Interpreter, this documentation was written with it, so it can be read confortably in a text editor (see `gates-doc.txt`).

At first sight, gates are just a convoluted way to define and execute macros. But they're better thought of as bricks to *construct* macros in such a way that each brick can be independantly controlled, and new bricks can be added to a wall already built. For each logical step in a big macro, the user can decide whether to execute that step, bypass it, add a new step before or after, etc. This way, you give the user the ability to tweak your macro as s/he wishes.

Suppose for instance you write `\BigMacro`. `\BigMacro` does a lot of interesting things : first it does A, then B, then – last but not least – C. The problem with `\BigMacro` is that it's very good ; so good that everybody uses it and wants to add his or her little twist to it, because `\BigMacro` really entices creativity. You have no time to add those twists yourself, nor do you think they'll interest anybody but their authors. Yet you keep receiving emails asking you to modify `\BigMacro`. Yes, `\BigMacro` is big, so most users don't care to try and understand it ; even if they do, they're not so sure how it works : in A, there seems to happen something, and X becomes Y, and then in B, Y is passed to… or was it X ? Plus those same people really don't want to copy one hundred lines of code just to add their touch to it. `\BigMacro` is like a plate of spaghetti : good, but spaghetti nonetheless, hard to decipher and rearrange. A few people will be able to hack it, but it is lost on users bold but lacking experience.

Gates are made to avoid that situation. `\BigMacro` will now be

made of the same logical steps, but they will be gates, which means users will be able to control their behavior without ever looking at how they're implemented. For instance, one person would prefer to avoid step B in some cases ; if B is a gate, that can be done easily. Another don't like B at all and wants to remove it altogether : nothing simpler. A third person would like to get the output of B and modify it a bit before C kicks in ; s/he just has to write a new gate (which is like writing a macro) and add it after B ; whatever B returns will be passed to that new gate instead of C, and whatever the new gate returns will be passed to C. That's another thing with gates : they pass arguments between them when one follows the other ; they can even return arguments. That's nothing new in Lua ; but in TEX, arguments handling is generally not so simple. In most, if not all, programming languages, the following :

```
dosomething(makesomething(<argument>))
```

is evaluated as follows : *<argument>* is passed to `makesomething`, whose return value is passed to `dosomething`. The equivalent in TEX doesn't do that, however :

```
\dosomething{\makesomething{<argument>}}
```

Here `\makesomething{<argument}` is passed to `\dosomething` and

chaos is sure to ensue if the previous behavior is expected. Instead, one must generally store whatever `\makesomething` 'returns' in a macro and pass that macro to `\dosomething`. With gates, however, the passing of arguments can be mimicked : if two gates follow each other in a list, whatever the first returns (no quotes here) is passed to the second. In our case, if `\makesomething` is called before `\dosomething`, the former will process its argument and pass its return value (if any) to the latter. No special care is needed to do so.

Another property of gates, obviously, is that they are reusable. In the TEX world, however, it is customary to distinguish between public macros, to be manipulated by users, and private ones, traditionally marked by an `@` in their names, and that lay users should absolutely ignore. With gates, everything seems public. That's indeed the very reason why gates exist : so that macros are as hackable as possible, and users can do what they like with them. If you're afraid that chaos might ensue (because users are notoriously dangerous), don't worry too much : it is a little bit harder to grasp how gates work than to turn `@` into a letter so it can be used in private macros ; a user who would manipulate gates thus probably knows what s/he's doing, or at least that what s/he's doing might break. (By the way, gates are very likely to contain `@`-marked private macros anyway, if only because they don't make sense outside the context of their use.)

If you want to use gates but still insist that users shouldn't mess with them, there is another solution : don't document them. Indeed, gates are useful only to the extent that other people know where they occur and what they do. So documenting gates is an essential step in exploiting them fully ; as will be said again below, though, there is no need to explain what a gate does internally : simply saying what arguments it takes, what it does with them (conceptually), and what it returns, is enough. In other words, you don't explain the code that makes up the gate, you explain its function in the larger picture.

One last remark about gates, not expanded in this document : they can be useful to create libraries, i.e. snippets of code to be used in various places. Instead of writing macros to perform such and such operation, you can write gates ; the difference is that they can be easily added to existing code, and externally controlled.

1.2 An overview of (almost) all actions

There are two types of gates : macro gates (or 'm-gates' for short) and list gates ('l-gates'). M-gates execute some code, like macros in TEX or functions in Lua ; l-gates contain other gates (of either type) and call them in turn, passing arguments between them ; from now on, 'gate' means a gate of either type. Typically, a big macro is an l-gate with many subgates, themselves possibly l-gates containing gates of a lower level still, and so on and so forth. Then one can say, 'In that l-gate, I want this gate to be ignored' ; or 'I want to add my own gate to this l-gate, in such position', etc.

Defining and executing gates To define an m-gate, one uses the `def` action ; the `list` action is used to declare an l-gate. Gates of either type can then be added to an l-gate with `add`; by default, the insertion is done at the end of the list, but a position relative to other gates can be optionally specified. Gates can be removed with the `remove` action.

A gate can also be defined by `copy`ing another gate ; in that case, the new gate is identical to the copied one, except its status (see below) is set to `open`, which is the default status when creating a gate. When copying an l-gate, the list is copied too (also it is the same subgates that occur in both l-gates).

To call a gate, one uses `execute` (various shorthands are possible). Executing an m-gate is not very different from executing a macro or function : it performs its definition. The execution of an l-gate, on the other hand, consists in calling the gates that were added to it. Also, and most importantly, gates in a l-gate pass arguments between them : the first gate receives the arguments passed to the l-gate it belongs to, then passes them to the next gate, possibly modifying some of those arguments by `return`ing. (Returning arguments depends heavily on the implementation ; see `return` and associates in TEX, and `autoreturn` in Lua.)

For each language, see : *defining and executing gates in TEX* and *defining and executing gates in Lua*.

Conditions Gates can be externally controlled in four ways. First, all gates have a `status`: if it is `open` (which is default), the gate is executed when encountered ; if it is `close`, it is ignored ; `ajar` means that the gate will be executed the next time it is encountered, and then its status will revert to `close`; `skip` is the opposite : the gate will be ignored the next time, then it will revert to `open`. In short, `ajar` and `skip` mean executing or ignoring the gate once. See *status in TEX* and *status in Lua*.

Second, all gates may have a `conditional`: if it is true, the gate is executed, and ignored otherwise. This allows gates to depend on external states of affairs. But a gate's conditional takes the same arguments as the gate itself, so the its value can also depend on what is passed to the gate. Thus, what a gate does and why it does it are kept distinct. See *conditional in TEX* and *conditional in Lua*.

Third, the same gate can be repeated with a `loop`: it is the same thing as a conditional, and the gate's arguments are also passed to it. The gate will be called repeatedly as long as the loop is true. When a gate is repeated, it passes its returned arguments to itself, and before that to the loop. It is as if the same gate had been added several times in a row to the same l-gate. See *loop in TEX* and *loop in Lua*.

Fourth, `loopuntil` is another kind of loop : the gate is repeated as long as it is false ; also, `loopuntil` is evaluated after the gate, so the latter is executed at least once. If a gate has both `loop` and

`loopuntil`, the latter is ignored. See *loop-until in TEX* and *loop-until in Lua*.

Finally, in Lua a gate may also have an `iterator`; to put it simply, it mimicks a `for` loop; for instance, a gate may be called with a table as its argument but actually be executed on every entry, if the `iterator` is the `pairs` function. See *iterator in Lua* for details.

Actually, a gate doesn't have only one status, one conditional, etc. Rather, it has one global set of conditions, and one local set of conditions for each l-gate where it appears (note that if a gate appears several time in the same l-gate, it has only one local set of conditions). The global conditions are examined whenever the gate is encountered, either when it is called directly with `execute` or when it is called by an l-gate; local conditions are examined in the latter case only.

It is important to understand the relative order of evaluation of the conditions, and how global and local conditions interact. Suppose a gate is called directly with `execute`; then the following happens: first, the gate's global status is checked; if `ajar` or `skip`, it is set back to `close` and `open` respectively; if the original status allows the execution of the gate (i.e. the status was `open` or `ajar`), the gate's global conditional is then evaluated; if it is true, and the gate has neither `loop` nor `loopuntil`, the gate is executed once; otherwise, it is repeated as long as `loop` is true or as long as `loopuntil` is false. Note that even if the gate is executed several

times, the status and conditional aren't reevaluated, only the loops. The same is true with `iterator` in Lua.

Suppose now the gate occurs in an l-gate. Then the same evaluation happens, this time with the local values for that given l-gate. If the gate is deemed good for execution, or rather, each time it is executed (if a loop calls it several time), the global values are evaluated again as we've just seen. In other words, when a gate occurs in an l-gate, its global conditions are examined if and only if the local conditions allows the execution of the gate, and each time they prompt it. This means for instance that if a gate is globally `ajar` and is encountered in an l-gate where it's local status is `close`, its global status will not be reverted to `open` because it will simply not be evaluated (of course, it can be evaluated elsewhere).

This may seem a bit complicated, but actually situations where you have to specify both local and global conditions are rare (global status is always a bit dangerous, because all the instances of the gate are affected, which might not be expected if somebody reuses an existing gate) ; also, you may very well `open` and `close` a gate for whatever reasons, but if you use `ajar` and `skip` it means you're controlling it quite tightly and you probably won't use a conditional too.

The shorthand notation Defining gates, adding them to l-gates, and setting their conditions, can be done with actions, as explained

above. But that can be tedious, because when you're building a big macro (a big l-gate, rather), it's hard to see the larger picture : you're adding gates one after the other, and you can't readily figure out what the big l-gate looks like. It's as if you were looking at a house brick by brick without being able to take a few steps back and consider the whole building.

But entire gates, replete with subgates down to whatever level, and specified for all conditions, can be created without ever mentioning any action. The overall architecture can be preserved, because gates are defined and added to an l-gate at once. It is a bit like writing a big macro or function, except you add a tag to each chunk of code, for further reference. The shorthand notation isn't explained further here, because it depends on the implementation and works very differently in TEX and Lua : see *TEX shorthand notation* and *Lua shorthand notation*.

Gate families Since gates are designed to be hacked, it is all the more convenient if they have simple, descriptive names. Of course, you can call a gate `insftn`, but `InsertFootnotes`, or any other readable name, is definitely better ; a user can then easily browse gates and find what s/he's looking for.

However, such significant names increase the risk that two gates bear the same name and clash. This is avoided by creating gate families. A family is simply a prefix added to a name's gate, so that

a gate whose name is `MyGate` is actually called `MyFamily:MyGate`. But family are associated with calling commands in TEX and tables in Lua, so that when a gate is mentioned without its family, it is automatically added. For instances the `\gates` command in TEX and the `gates` table in Lua are associated with the family called `gates`, so the name of a gate manipulated with them is actually `gates:<name>`.

You can define a new family, associated with a command or table, with the `new` action ; gates manipulated by that command or table will then have a real name which includes the family. Thus, their apparent names can be identical to other gates, as long as they belong to other families.

As said above, a gate's family is supplied when it is missing from the gate's name. This means conversely that if the family is specified in a gate's name, it is not added. So, if you mention gate `MyFam:MyGate`, the family associated with the command or table where the gate is mentioned isn't specified again. You can thus very well use gates from other families without having to rely on the associated command or table. In other words, there's no hidden mechanism behind families : they're just prefixes added to a gate's name, and the automatic addition of a family is just an examination of a gate's name.

Although a command or table must be declared with `new` to work properly, families themselves don't require that. Gate `MyFam:MyGate`

can be used even if `MyFam` hasn't been declared (and the declaration can also take place later, so that the family is associated with a command or table). In other words, the command/table-family association is just a convenient way to make the names of your gates unique without thinking about it, but you can also think about it and use an explicit family prefix. See *families in TEX* and *families in Lua*

If you get lost Gates can be quite complex. For instance, the big l-gate that creates section headings in this document is made of 16 subgates and subsubgates (as I am writing this, at least). The main function of the Interpreter package used to turn the source of this document into proper TEX is made of more than 25 subgates (not counting repeated ones), with 7 levels between the top l-gate and the most deeply embedded m-gate.

There are a few actions to make things clearer. First, you can know whether a given gate is an m- or l-gate with `type`, which returns 1 (for m-gates), 2 (for l-gates), or 0 (if the name you passed isn't a gate). Similarly, a gate's status (either global or local to some l-gate) can be queried with the `status` action, which returns 1, 2, 3 or 4, depending on whether the gate's status is open, ajar, skip or close, and 0 if there is no gate with the given name.

You can loop over all the subgates in an l-gate with `subgates`, and execute some code for each subgate.

The family associated with a command or table can be queried by the `family` action.

Finally, gates can be explored more thoroughly with `show` and `trace`. The `show` action prints (on the terminal and log file) a gate's name, type, and global conditions (the loops are omitted if not specified) ; if the gate is an l-gate, the same happens with its subgates, except local conditions are shown ; and a subgate is an l-gate itself, the process goes on. Subgates are displayed in a manner similar to the TEX shorthand notation.

The `trace` action shows gates when they are encountered : it is signalled whether they are executed or not (and why), and possibly the arguments passed to them are mentioned too. Again, subgates to an l-gate are marked as such.

See *getting lost in TEX* and *getting lost in Lua*

Gates in TEX

This part of the documentation explains how gates work in TEX.

The way to load Gates depends on your format. In plain TEX, you use :

2.1 Loading and using Gates

```
\input gates
```

In LaTEX :

```
\usepackage{gates}
```

And in ConTEXt:

```
\usemodule[gates]
```

Actions in TEX aren't executed with control sequences, but with a calling command (by default, `\gates`), followed by an action's name, followed by a space:

```
\gates action {...}
```

(Of course, the space can be the end of a line.)

Note that loading Gates in TEX doesn't automatically load the Lua counterpart even in LuaTEX; in other words, Gates in Lua should be independantly loaded.

Here it is shown how gates are created, concatenated, executed, and how arguments are passed between them.

2.2 Definition and execution

Defining gates Let's try an extremely simple example. You want to define a macro which, when given a number, adds 3 to it, multiplies everything by 2, and prints the result. How fascinating. Here's how you'd do it with gates. First you define your m-gates with `def`:

```
\gates def {add} [1] {%
  \gates return {#1+3}
  }
\gates def {multiply} [1] {%
  \gates return {(#1)*2}
  }
\gates def {print} [1] {%
  \the\numexpr#1\relax
  }
```

The number of argument (up to nine) is given after the name, between brackets, and can be omitted if the gate takes no argument. The `return` action is discussed more thoroughly *below*; just note for the moment that there is no need to add a comment at the end of the line to avoid spurious space, because material after a `return` is gobbled (so watch your `\fi`'s !).

(Instead of `def`, we could use the `edef` action ; the difference is the same as between TEX's `\def` and `\edef`. There is no `\gdef`/`\xdef` variants, because operations on gates are always globals.)

Now those gates should be added to an l-gate. Such a gate is declared with `list`, and an optional number of arguments can be specified too.

```
\gates list {operation} [1]
```

Finally we add the m-gates to the l-gate :

```
\gates add {add, multiply, print} {operation}
```

The add action can take one or more gates, separated by commas. Examples of add with specified position can be found *below*.

Executing gates And that's it, our small \BigMacro is done. We can call it, with 4 as argument, for instance, and it will print 14 ; to do so, we use the execute action :

```
\gates execute {operation}{4}
```

Note that m-gates can be executed too, although *that* is a convoluted way to call a macro ; in this case, do not worry about the return value, it simply vanishes. For instance, the following :

```
\gates execute {print}{5+2}
```

will produce 7. If we'd call multiply rather than print, nothing would have happened, since it simply returns something, which doesn't make sense outside an l-gate.

Execution can be called more directly as :

```
\gates operation {4}
```

I.e. instead of a gate action, you use the name of a gate, and that's equivalent to calling `execute` with that gate. However, this can be done if and only if the gate you want to call doesn't have the same name as an action; for instance, you can't do that with the `add` gate, because there exists an action called `add`, so you have to use `execute` instead.

Now suppose we'd like to add a substraction after the multiplication – that is, suppose we're a user who wants to add his or her touch to `\BigMacro`. Nothing simpler:

```
\gates def {substract} [1] {%
  \gates return {#1-3}
  }
\gates add {substract}[after multiply]{operation}
```

And now `operation` returns `11` when fed 4. This was done by simply adding an optional argument to the `add` operation; this specifies where the new gate(s) should be added in the l-gate: by default, it is the end of the list, but you can say `first` to put the gate(s) at the beginning or `before <name>` or `after <name>` to make the insertion before or after the gate called *<name>*; instead of a name, you can also use `first` or `last` to denote the first and last gates of

the list, so here we could have used `before last` instead of `after multiply`.

You can also `remove` a gate from a list. After:

```
\gates remove {substract}{operation}
```

the l-gate `operation` is now the same as before.

Handling arguments One crucial properties of gates is that they pass arguments between them. This was already illustrated by our code above: `operation` takes one argument, which is passed to `add`, `multiply` and `print`; or rather, it is passed to `add`, and that gate returns an argument which is passed to `multiply`, and again to `print`.

Arguments work as follows: a gate called with `execute` should be followed by as many arguments as it was declared with, just like any other macro in TEX. An l-gate passes its arguments to its first subgate; and for two consecutive gates, the second receives what the first returns.

To return arguments, you use the `return` action, as illustrated above. It expects the same number of arguments as the gate where it appears. For instance, `add` was declared with one argument, so when it calls `return`, one argument is expected. But one might want to return a different number of arguments; in this case, one

can use `return0`, `return1`, `return2`, etc., up to `return9`. Also, the `return!` action takes one big argument containing an indefinite number of arguments to be returned ; for instance :

```
\gates return2 {one}{two}
\gates return! {{one}{two}}
```

are equivalent, and they're also equivalent to a simple `return` with two arguments in a gate declared as taking two arguments.

Since l-gates execute no code, you can't call any version of `return` with them ; but they automatically return whatever was returned by their last gate, and that is passed whatever follows the l-gate.

Until now, we have implicitly assumed that all gates in the same l-gate take the same number of arguments. That is not necessary : two gates with a different number of arguments can occur in the same l-gate ; consequently, a gate doesn't need to take the same number of arguments as the l-gate it appears in. For instance, you can very well declare an l-gate with 3 arguments, and add to it an m-gate with 2 arguments and another one with 4. Let's do it, and see what happens :

```
\gates list {mylist} [3]
\gates def  {macro1} [2] {%
  \immediate\write16{1: #1. 2: #2.}}
```

```
\gates def  {macro2} [4] {%
  \immediate\write16{1: #1. 2: #2. 3: #3. 4: #4.}}
\gates add {macro1, macro2} {mylist}
\gates execute {mylist} {one}{two}{three}
```

This will print :

```
1: one. 2: two
1: one. 2: two. 3: three. 4: .
```

The first two arguments of `mylist` are passed to `macro1`, and the third is simply ignored. Then the three arguments are passed to `macro2`, along with a empty fourth one. In other words, the number of arguments is adjusted so that every gate receives what it needs and nothing more.

But haven't we said just above that a gate receives what the previous one returns ? Since `macro1` returns nothing, shouldn't `macro2` receive four empty arguments ? Things are actually a bit more complicated to explain, but simpler to use : given two consecutive gates A and B, the arguments passed to B are whatever A returns, plus additional arguments taken from those that A received if necessary (i.e. if B takes more arguments than A has returned). To put it differently, when a gate returns *n* arguments, they simply replace the first *n* arguments of the current arguments, which are

passed to the next gate. In our example, macro1 returned nothing, so the arguments that it received were restored and passed to macro2. Here is another example:

```
\gates list {mylist} [3]
\gates def  {macro1} [2] {%
  \immediate\write16{1: #1. 2: #2.}%
  \gates return1 {ONE}}
\gates def  {macro2} [4] {%
  \immediate\write16{1: #1. 2: #2. 3: #3. 4: #4.}%
  \gates return {first}{second}{third}{fourth}}
\gates def {macro3} [4] {%
  \immediate\write16{1: #1. 2: #2. 3: #3. 4: #4.}}
\gates add {macro1, macro2, macro3} {mylist}
\gates execute {mylist} {one}{two}{three}
```

And this prints:

```
1: one. 2: two.
1: ONE. 2: two. 3: three. 4: .
1: first. 2: second. 3: third. 4: fourth.
```

It works as follows: macro1 uses the first two arguments from mylist, and returns one; macro2 thus takes that argument, plus

the second, third and fourth original arguments (the fourth being empty, since `mylist` takes only three) ; four argument are returned, which is all `macro3` needs, so there is no empty argument.

As said above, l-gates automatically return what their last gates return ; more accurately, an l-gate declared with *n* arguments returns the same number of arguments, those being taken from what the last gate returns plus additional arguments restored as we've just seen. In our example, `mylist` will return `first`, `second` and `third` (it takes, and thus returns, three arguments), because `macro3` returned nothing and the arguments that it received from `macro2` are restored. (Actually, returning doesn't mean anything here because `mylist` doesn't appear in a l-gate itself.)

This behavior is quite convenient, because it means that when a gate doesn't modify an argument, it doesn't have to bother to return it.

2.3 Controlling gates

Here it is explained how to change a gate's conditions.

Status The global status of a gate (i.e. its behavior wherever it is encountered) can be set with the following actions : `open!`, `ajar!`, `skip!` and `close!`, as in :

```
\gates ajar! {mygate, myothergate}
```

This also illustrates that you can set the status of several gates at once, separating them with commas (spaces are ignored).

The local status (i.e. the behavior of a gate in a given l-gate) is set with the same action without an exclamation mark, followed by the l-gate where you want to specify the status :

```
\gates open {mygate, myothergate}{mylist}
```

But you can also use `before <gate>` and `after <gate>`, meaning that the status of all gates before (resp. after) *<gate>* in the l-gate will be set accordingly. For instance :

```
\gates close {before mygate}{mylist}
```

As an example of a status-controlled gate, let's consider the problem of automatically removing the indentation of a paragraph. The usual solution is :

```
\everypar={{\setbox0=\lastbox}}
```

The `\everypar` token list is inserted every time TEX begins a paragraph ; assigning `\lastbox` to a box register removes it from the list under construction : here the indentation box is thus removed from the paragraph ; the assignment is done in a group

so box 0 remains unaffected. The problem is that other resources might want to use `\everypar`; if the exploitation of the token list is always so direct, one resource might wipe another. With gates, the solution is to plant an l-gate in `\everypar`; then you can add whatever gate to it without disturbing the other gates possibly there too:

```
\gates list {everypar}
\gates def  {noindent}{{\setbox0=\lastbox}}
\gates add  {noindent}{everypar}
\everypar={\gates execute {everypar}}
```

But we were interested in status. As is, all paragraphs will be unindented, because `noindent` will be executed for each paragraph. Instead, we should close it with

```
\gates close {noindent}{everypar}
```

and situations triggering an unindented paragraph (for instance, a section header) should call

```
\gates ajar {noindent}{everypar}
```

so that `noindent` will make its job once, and then close.

Conditional Status is useful, but it suffers one flaw: you have to set it yourself. That's no problem when you're handling a local situation like the previous one; but things might a be a bit more far-reaching. You might want X to influence Y, and X and Y might not be related at all; also, X might change more than once.

For more flexible control, gates can depend on a `conditional`, for instance (illustrating both global and local setting):

```
\gates conditional! {mygate}{\ifSomething}
\gates conditional  {gate1, gate2}{mylist}{\ifSomethingElse}
```

Now, `mygate` will be executed only when `\ifSomething` is true, just like `gate1` and `gate2` in `mylist` will be executed in `mylist` when `\ifSomethingElse` is true. The situations where conditionals can be put to use are countless. A preface in a book, for instance, will certainly make some `\ifSpecialChapter` conditional true, and many details will depend on that conditional: e.g. unnumbered section headings, page number in roman numerals, etc. If those are implemented with gates, setting `\ifSpecialChapter` as the conditional for those gates will make them depend on the larger picture (you could use status too, but the conditional is more powerful because many different things can depend on it, not only gates).

What constitutes a valid conditional? Technically, whatever

fits *texapi*'s `\ifexpression`. Indeed, Gates is written with *texapi*, and conditionals rely internally on `\ifexpression`. What then fits `\ifexpression`? First, any traditional TEX conditional, no matter whether it's a primitive like `\ifhmode` or `\ifcat` or a macro defined with `\newif`. Second, argument-taking conditionals, i.e. macros working like

```
\ifXXX {<true>}{<false>}
```

can be used too (*texapi* itself defines a good deal of such conditionals). In both cases, what should be set in `conditional` is the test itself, not the *<true>* and *<false>* parts. For instance, with `\ifSomething` and `\ifSomethingElse` above, the true branch and the `\else ... \fi` continuation were left out of the picture. If you used, say, `\ifnum`, you'd specify something like (note the necessary space):

```
\gates conditional {mygate}{mylist}{\ifnum0=1 }
```

With a *texapi* conditional you could use:

```
\gates conditional {mygate}{mylist}{\ifstring{foo}{bar}}
```

(Here of course we have specified stupid conditionals which are always false; more useful examples can be found below).

But `\ifexpression` (hence `conditional`) also allows using simple logical operators : `&` means *and*, `|` means *or*, `-` means *not*, and subexpressions can be created with braces. For instance, the following two conditionals are equivalent and are true if the absolute value of N is smaller than 100 (the space following an operator is ignored) :

```
\ifnum N > -100 & \ifnum N < 100
-{\ifnum N < -99 | \ifnum N > 99 }
```

Pretty complex expressions can thus be assigned to a gate's conditional. But things are better yet : conditionals take arguments, the same as the gates they control, so that the test can depend not only on external conditions but on what a gate receives. For instance, with :

```
\gates conditional {mygate}{mylist}{\ifnum#1>0 }
```

`mygate` will be executed only when its first argument is positive. Another example :

```
\gates conditional! {mygate}{-\ifstring{#1}{#2}}
```

Here, `mygate` will be executed only when its first two arguments

differ, no matter in which list `mygate` appear, since the global conditional is set here.

If you don't want a gate to depend on a conditional anymore, simply declare something like :

```
\gates conditional! {mygate}{\iftrue}
```

Loop Third type of control : the execution of a gate can be repeated as long as some condition is true. To do so, you use the `loop` action (with an exclamation mark if you want to set the global loop). It takes the same stuff as `conditional`, and the gate's arguments are passed to it too. If the material evaluates to true, the gate is executed, then the material is reevaluated, and if it is true again the gate is executed again, and so on and so forth, so obviously something must happen so that the iteration stops.

When a gate repeats the arguments it returns are passed back to itself, and before that to the loop. Here is an example :

```
\gates def {myloop} [1] {%
  \immediate\write16{\the\numexpr#1}%
  \gates return {#1+1}}
\gates loop! {myloop} {\ifnum\numexpr#1<5 }
\gates execute {myloop}{1}
```

This will print numbers from 1 to 4. Once 4 is printed, `myloop` returns 5 (1+1+1+1+1, really), so the conditional controlling the loop isn't true anymore and the iteration stops. If this happened in an l-gate, the return value of `myloop` would be passed to the next gate.

When a gate shouldn't be controlled by a loop anymore, you can use `noloop` (with or without an exclamation mark):

```
\gates noloop! {mygate}
```

Loop until You can specify another type of loop with `loopuntil`: it works like `loop`, except the gate is repeated as long as the material evaluates to false. Also, the evaluation takes place after the gate is executed, which means that the gate is always executed at least once; arguments are repeatedly passed to the gate and the conditional as with `loop`, except `loopuntil` always receives the return values of the gate it controls (whereas on the first evaluation `loop` uses the arguments passed to gate, not what it returns). Here's the previous gate rewritten with `loopuntil`; it will work slightly differently:

```
\gates def {myloop} [1] {%
  \immediate\write16{\the\numexpr#1}%
  \gates return {#1+1}}
```

```
\gates loopuntil! {myloop} {\ifnum\numexpr#1>4 }
\gates execute {myloop}{1}
```

(The difference with the previous version is that here `myloop` will be executed at least once, even if a number larger than 4 is passed to it. Also, it is important to use `\ifnum\numexpr#1>4` and not `\ifnum\numexpr#1=5` ; the latter will work if `myloop` is always executed with a number smaller than 5, but it will enter an infinite loop otherwise, since the condition will never be true.)

If you want a gate to stop being controlled by `loopuntil`, use `noloopuntil`.

Note that if both `loop` and `loopuntil` are specified, the latter is ignored.

2.4 The shorthand notation

The shorthand notation allows you to define gates (e.g. l-gates with subgates), including conditions, in one go, without having to call any gate actions. This makes complex code much more readable.

We've already seen the implicit call to `execute` by using a gate's name instead of an action. You can also define m- and l- gates more directly as follows :

```
\gates def  {mygate} [2] {...}
\gates list {mylist} [2]
```

are equivalent to

```
\gates [mygate] [2] {...}
\gates (mylist) [2]
```

and the gate's name can be surrounded by space, it is trimmed away. (The examples in this section all use this shorthand, but they would be still valid with the explicit `def` and `list` instead.) Second, following the number of arguments (or the gate's name if the gate takes no argument), you can optionally specify global conditions by using `?` followed by a list of `key = value` pairs, where the keys are `status`, `conditional`, `loop` and `loopuntil`; for `status`, the value should be one of `open`, `ajar`, `skip` or `close`, and for the other conditions it should be a conditional as seen above. For instance, here's the creation of an m-gate with status `close` and some loop:

```
\gates [mygate] [2] ?{status = close,
                      loop   = \ifnum#1<5 } {...}
```

This is equivalent to:

```
\gates [mygate] [2] {...}
\gates close! {mygate}
```

```
\gates loop!  {mygate}{\ifnum#1<5 }
```

Note that the key is trimmed, as is the value if the key is `status`, so space can be used to make things readable. In the case of a conditional or loop's value, it isn't trimmed, because space can be important, as in our example here where it delimits the number `5`. The space on the left is insignificant, though.

And here comes the most interesting shorthand : when declaring an l-gate, you can define and add subgates at once by declaring them just after the l-gate (and its optional number of arguments and/or conditions, if any), using the same notations as with implicit definitions : `[mygate]` or `(mylist)`, optionally followed by the number of arguments and/or the conditions, and with a definition in case of an m-gate. The only difference is that the conditions thus specified, if any, are the local ones relative to the l-gate under construction. In other words, the following :

```
\gates (mylist) [2]
   [mygate] [2] ?{status = ajar} {...}
   (myotherlist) [1]?{loop = \ifnum#1<5 }
```

is equivalent to the much more verbose

```
\gates (mylist) [2]
```

```
\gates [mygate] [2] {...}
\gates (myotherlist) [1]
\gates add {mygate, myotherlist}{mylist}
\gates ajar {mygate}{mylist}
\gates loop {myotherlist}{mylist}{\ifnum#1<5 }
```

The shorthand notation allows you to see at once that `mylist` contains `mygate` and `myotherlist`. But then, the latter is an l-gate too, so it could be nice if we could add subgates to it in the same way; well, we can, it suffices to use a dot:

```
\gates (mylist) [2]
   [mygate] [2] ?{status = ajar} {...}
   (myotherlist) [1] ?{loop = \ifnum#1<5 }
   . [mysubgate] [1]?{...} {...}
```

Here `mysubgate` will be added to `myotherlist` instead of `mylist`, and the conditions, if any, will be the local conditions relative to `myotherlist`, not `mylist`. Now what if `myotherlist` contains an l-gate `mysublist` and you want to add subgates to that one? Well, you use two dots, and so on and so forth (I leave argument numbers and conditions aside to make things simpler):

```
\gates (mylist)
```

```
[mygate] {...}
(myotherlist)
.  [mysubgate] {...}
.  (mysublist)
.  .  [mysubsubgate] {...}
.  .  (yetanotherlist)
.  .  .  (pleasestop)
```

etc., etc. The notation with dots work as follows : if we assume that gates are added to l-gates of a given level, then dots denote that level : no dot means that the gate is added to the l-gate of level 0, i.e. the gate defined with `list` (`mylist` here), and is itself of level 1, one dot means that the gate is added to an l-gate of level 1 and is itself of level 2 (e.g. `mysubgate` added to `myotherlist`), and so on and so forth. Space around the dot is insignificant. The only (obvious) restriction is that an l-gate at level *n* isn't available anymore if followed by a gate at the same level or lower. This works a bit like a table of contents : a subsection can't be added if there isn't a section immediately above it. If you want to come back to a gate at another level, just use less dots. For instance, if the code above continued with

```
.  .  [gateX] {...}
[gateY] {...}
```

then gateX and gateY would be defined and added to mysublist and mylist respectively. Of course, after gateX, yetanotherlist and pleasestop aren't available for insertion anymore ; similarly, after gateY, only mylist can host incoming gates, until a new l-gate is declared.

It's nice to define subgates and add them to an l-gate at once, but sometimes you might want to add a gate that already exists. Using the above notation would redefine it. Of course, you can still add and specify the position, but that's not very convenient. So, besides parentheses and square brackets, you can use <mygate> to add mygate without redefining it ; you can actually add several gates at once, by separating them with commas. Local conditions can then be set as above. For instance :

```
\gates (mylist)
  [gateA] {...}
  <gateB, gateC> ?{status = close}
  (myotherlist)
  . <gateD>
```

creates l-gate mylist with subgates gateA, gateB, gateC and myotherlist, gateB and gateC being locally closed, and myotherlist containing gateD. This of course requires that gateB, gateC and gateD already exist. If a gate thus added is an l-gate, the dot notation can be used to add subgates to it.

The dot is actually only the default character to signal subgates. You can use others by declaring them with `subchar`, as in :

```
\gates subchar #
```

The character thus declared do not replace existing ones (this would be dangerous) but simply adds a new possibility. For obvious reason, the character can't be `[`, `(`, `<` or `?`.

New gate families can be created with the `new` action.

2.5 Gate families

```
\gates new \MyGates {MyFamily}
```

Now `\MyGates` will work exactly like `\gates`, except that when it manipulates gates, they will all have the prefix `MyFamily:` attached to them. For instance :

```
\gates def {mygate}{...}
\MyGates def {mygate}{...}
```

defines two different gates, `gates:mygate` and `MyFamily:mygate`.

If a family is explicitly given in a gate's name, the family associated with the calling command isn't added. The following two lines are equivalent, for instance :

```
\gates def {MyFamily:mygate}{...}
\MyGates def {mygate}{...}
```

Of course the shorthand notation allows you to mix families:

```
\MyGates (MyList)
  (MySubList)
  . [MyGate] {...}
  . [AnotherFamily:AnotherGate] {...}
```

The family associated with a command can be queried with the `family` action. For instance:

```
\MyGates family
```

returns `MyFamily`.

Both `type` and `status` return a number: `type` returns 0 if there is no gate with that name, 1 if it is an m-gate and 2 for an l-gate; `status` returns 0 for a non-existing gate, and 1 to 4 for `open`, `ajar`, `skip` and `close` respectively; `status` can also be used with an exclamation mark to query global status:

2.6 If you get lost

```
\gates status  {mygate}{mylist}
```

```
\gates status! {mygate}
```

The subgates action takes a definition as its second argument, which will be executed with #1 replaced with the subgate's name. For instance, the following shows all the subgates in mylist, with their types:

```
\gates subgates {mylist}{%
  \immediate\write16{%
    #1 \ifnum\gates type {#1}=1
      (m-gate)\else (l-gate)\fi}%
   }
```

The show action can be used to display a gate's construction and conditions. It is displayed as the shorthand notation. The trace action takes a number and works as follows: if the number is 0, gates of the associated families aren't traced; if it is 1, it is mentionned if they are called (in an l-gate) and executed or not, and why; with 2, it works like 1, except the arguments passed to the gates are shown too.

All actions are called with \gates, or any other calling command created with new, as follows:

\gates *<action><space>*

Executes *<action>*; if there is a gate called *<action>*, this is equivalent to \gates execute {<action>}. The name of the action (or gate) is delimited by a space; depending on *<action>*, arguments might be required.

In what follows, *<gate list>* denotes a comma-separated list of gates, e.g. mygate, myothergate, thirdgate; *<gate spec>* denotes either a *<gate list>* or a relative position of the form before <gate> or after <gate>

add *<gate list>*[*<position>*]*<l-gate>*

This adds all the gates in *<gate list>* to *<l-gate>*. If *<position>* is missing, the addition occurs at the end of the l-gate. Otherwise, it should be one of the following: first, meaning the gates will be added at the beginning of the l-gate; last, meaning they will be added at the end (so this is similar to no *<position>*); before <name>, and the addition will occur before gate *<name>* in the l-gate (it should of course exist); after <name>, and the addition will occur after gate *<name>*. Note that *<name>* can be first or last, denoting the first and last gates in *<l-gate>* (before first and after last are obviously synonymous with first and last).

ajar *<gate spec><l-gate>*

Sets the local status in *<l-gate>* to ajar for the gates in *<gate spec>*.

`ajar!` *<gate spec>*

Sets the global status for the gates in *<gate spec>* to `ajar`.

`close` *<gate spec><l-gate>*

Sets the local status in *<l-gate>* to `close` for the gates in *<gate spec>*.

`close!` *<gate spec>*

Sets the global status for the gates in *<gate spec>* to `close`.

`conditional` *<gate spec><l-gate><conditional>*

Sets the local conditional in *<l-gate>* to *<conditional>* for the gates in *<gate spec>*. The conditional should be anything that fits *texapi*'s `\ifexpression`.

`conditional!` *<gate spec><conditional>*

Sets the global conditional for the gates in *<gate spec>* to *<conditional>*.

`copy` *<gate1><gate2>*

Defines *<gate1>* as *<gate2>* (either an m- or l-gate). The current global status is also copied. If *<gate2>* is an l-gate, its gate list is also copied, along with the current status of the gates it contains. On the other hand, if *<gate2>* occurs in some l-gate(s), this information isn't copied.

`def` *<name>*`[`*<number of arguments>*`]`*<optional conditions>*`{`*<definition>*`}`

Defines *<name>* as an m-gate with the given number of arguments and definition. Such an assignment is always global. The number of arguments, ranging from 0 to 9, is optional, in which case the

gate takes no argument. In other words, the following lines are equivalent :

```
\gates def {mygate}{<definition>}
\gates def {mygate}[0]{<definition>}
```

Global conditions can be specified with ? followed by a key-value list, where a key is one of `status`, `conditional`, `loop` and `loopuntil` and the value is whatever fits the condition, as when explicitely calling the associated action with an exclamation mark. For instance,

```
\gates def {mygate}?{conditional = \ifsomething} {...}
```

is equivalent to

```
\gates def {mygate}{...}
\gates conditional! {mygate}{\ifsomething}
```

In the key-value list, the key is always trimmed of surrounding spaces, as is the value if the key is `status` ; values aren't trimmed for other conditions, because space might be significant (e.g. with `\ifnum`), but leading space is harmless (it is ignored by the internal processing of the conditional).

Giving an gate's name between brackets instead of an action is similar to using `def` with that gate :

```
\gates [mygate] [1] {...}
```

`edef` *<name>*`[`*<number of arguments>*`]`*<optional conditions>*`{`*<definition>*`}`
: Similar to def, but performs `\edef`.

`execute` *<name>*`<`*arguments*`>`
: Executes gate *<name>*. The number of *<arguments>* should match what *<name>* was defined with. If *<name>* is an m-gate, this is but a convoluted way of calling a macro. If *<name>* is an l-gate, this will launch sub-gates. Not that the execution depends on the gate's global conditions. Using a gate's name as an action is similar to using `execute` with that gate.

`family`
: Returns the family associated with the calling command.

`list` *<name>*`[`*<number of arguments>*`]`*<optional conditions>*`<`*optional subgates*`>`
: Defines *<name>* as an l-gate with the given number of arguments (can be missing if 0). The optional global conditions can be specified with `?` followed by a key-value list ; see `def` above. The *<optional subgates>* constitute the shorthand notation explained *above*, defining and adding subgates to the l-gate under construction, and specifying the local conditions too.

Giving an gate's name between parentheses instead of an action is similar to using `def` with that gate :

```
\gates (mylist) [2]?{status = ajar}
```

`loop` *<gate spec><1-gate><conditional>*

Sets the local while-loop in *<1-gate>* to *<conditional>* for the gates in *<gate spec>*. The conditional is the same as with `conditional`; the difference is that the gates will be executed again as long as the conditional is true, so there'd better be something somewhere which makes it false.

`loop!` *<gate spec><conditional>*.

Sets the global while-loop for the gates in *<gate spec>* to *<conditional>*.

`loopuntil` *<gate spec><1-gate><conditional>*

Sets the local until-loop in *<1-gate>* to *<conditional>* for the gates in *<gate spec>*. The conditional is the same as with `conditional`; the gates will be executed until the conditional is true; this means they will be executed at least once. If `loop` is also specified, `loopuntil` is ignored.

`loopuntil!` *<gate spec><conditional>*.

Sets the global until-loop for the gates in *<gate spec>* to *<conditional>*.

`new` *<control sequence><family>*

Defines *<control sequence>* as a calling command for gates, associated with *<family>*.

`noloop` *<gate spec><1-gate><conditional>*

Unsets the local while-loop in *<1-gate>* to *<conditional>* for the gates in *<gate spec>*.

`noloop!` *<gate spec>*

Unsets the global while-loop for the gates in *<gate spec>*.

`noloopuntil` *<gate spec><l-gate><conditional>*

Unsets the local until-loop in *<l-gate>* to *<conditional>* for the gates in *<gate spec>*.

`noloopuntil!` *<gate spec>*

Unsets the global until-loop for the gates in *<gate spec>*.

`open` *<gate spec><l-gate>*

Sets the local status in *<l-gate>* to open for the gates in *<gate spec>*.

`open!` *<gate spec>*

Sets the global status for the gates in *<gate spec>* to open.

`remove` *<gate list><l-gate>*

Removes the gates in *<gate list>* (names separated by commas) in *<l-gate>*.

`return` *<arguments>*

In an m-gate, pass *<arguments>* to the next one. There should be as many *<arguments>* as the gate was declared with. Any material following the statement in the gate's definition will be gobbled.

`return0, return1, return2 ... return8, return9`

Makes the gate return the specified number of arguments, no matter the number of arguments the gate was defined with.

`return!` *<arguments>*

Makes the gate return an indefinite number of arguments. If

<arguments> is empty, it is similar to 'return0'; if *<arguments>* contains one argument, it is similar to 'return1'; if it contains two, it is similar to 'return2', etc. An argument is defined as usual in TEX: a token, or balanced text. For instance, in what follows the second and third lines are equivalent; the first does the same job if and only if called inside a gate defined with three arguments (otherwise chaos will ensue).

```
\gates return  {one}{two}{three}
\gates return3 {one}{two}{three}
\gates return! {{one}{two}{three}}
```

show *<gate>*

Writes to the log and terminal *<gate>*'s type, its number of arguments, global status, conditional, loops (if specified), and recursively its subgates if *<gate>* is an l-gate (showing the local conditions).

skip *<gate spec><l-gate>*

Sets the local status in *<l-gate>* to `skip` for the gates in *<gate spec>*.

skip! *<gate spec>*

Sets the global status for the gates in *<gate spec>* to `skip`.

status *<gate><l-gate>*

Returns the local status of *<gate>* in *<l-gate>*: 1 if the gate is open, 2 if it is ajar, 3 if it is to be skipped, and 4 if it is closed; if

there is no *<gate>* in *<l-gate>* for whatever reason (including if there exists no such l-gate), returns 0.

`status!` *<gate>*
: Returns the global status of *<gate>*.

`subchar` *<character>*
: Defines *<character>* as denoting a subgate in the shorthand notation. `[`, `(`, `<` and `?` are forbidden.

`subgates` *<l-gate>**<definition>*
: Executes *<definition>* with each gate in *<l-gate>*. In *<definition>*, `#1` stands for the name of the subgate on each iteration.

`type` *<name>*
: Returns 0 if *<name>* is not a gate, 1 if it is an m-gate and 2 if it is an l-gate.

`trace` *<number>*
: If *<number>* is 0, gate operations aren't reported ; if 1, encountered gates are reported, along with their status and conditional when necessary ; if 2, arguments passed to gates that are executed are also displayed. Tracing affects only gates of the family associated with the calling command.

Gates in Lua

In general, gates behave similarly in TEX and Lua, but the syntax obviously differs. Also, there is one situation where both implementations diverge : when arguments are returned.

Gates in Lua aren't loaded automatically with the gates package. So one of the following should be issued somewhere :

```
dofile("gates.lua")
require("gates.lua")
```

Actually `dofile` requires more precision, e.g. (in LuaTEX) :

```
dofile(kpse.find_file("gates.lua"))
```

On the other hand, `require` is a little bit more clever (in LuaTEX, it uses `kpathsea`). Note that even if `dofile` is used, `gates.lua` won't be loaded twice, because the file returns at once if `gates` already exists.

In the above paragraphs, 'in LuaTEX' was mentioned twice : that's because Lua gates can be used in any Lua interpreter. Lua is obviously required, but not Lua*TEX*. In other words, Lua Gates don't rely on any special feature of Lua in LuaTEX (libraries, in particular, aren't used).

The file `gates.lua` returns nothing when loaded ; it simply creates the `gates` table, in which everything takes place until a `new` one is created with a family. This means that all actions are fields of the `gates` table.

Here's how to define and execute gates in Lua.

Defining gates The `def` action creates an m-gate; it takes a table with (for now) two entries: the entry at index `1` is the gate's name, the entry at index `2` is the function performed by the gate. For instance, we can translate our simple TEX example in Lua (since the table is the only argument to `def`, the surrounding parentheses can be removed):

```
gates.def {"add", function (n) return n + 3 end}
gates.def {"multiply", function (n) return n *2 end}
gates.def {"print", print}
```

The function can be either an anonymous function created on the fly, or a function variable, as with `print`. However, such a syntax is cumbersome in Lua, so you can directly assign to an entry in the `gates` table, provided it hasn't the same name as an action (so it is impossible with the `add` gate, for instance):

```
function gates.multiply (n)
  return n * 2
end
```

Actually, entries in the `gates` table be assigned any type; if a

function is assigned, a gate is created; otherwise, `gates` behaves as an ordinary table with an ordinary entry:

```
gates.mystring = "Hello"
print(gates.mystring)
```

However, the entry can't be redefined as a gate anymore; if it is assigned a function, it will be nothing more. In the following code, `bar` is a gate, but `foo` isn't.

```
gates.bar = function () ... end
gates.foo = "hello"
gates.foo = function () ... end
```

But let's get back to proper gates. We want to add our m-gates to an l-gate, which we declare beforehand with `list`:

```
gates.list {"operation"}
```

Like `def`, `list` takes a table as its single argument; the only required field is the gate's name at index `1`. We can now add the m-gates to the l-gate:

```
gates.add ({"add", "multiply", "print"}, "operation")
```

The add action takes a table containing subgates as its first argument, and a string representing the l-gate where the insertion is done as its second. If you add a single subgate, the first argument can be a string. More on add *below*.

Executing gates We can now execute our l-gate :

```
gates.execute ("operation", 4)
```

and 14 will be printed on the terminal. The execute action takes a gate as its first arguments, and then the arguments that are to be passed to the gate. Again, this syntax is a bit cumbersome, and gates can be executed more naturally as :

```
gates.execute.operation(4)
```

or simpler, but provided the gate doesn't share a name with an action :

```
gates.operation(4)
```

The last two variants also let you retrieve the gate itself as a function, instead of calling it :

```
x = gates.execute.operation
callback.register("process_input_buffer", gates.operation)
```

Now we can define another m-gate and add it to `operation`, specifying the position:

```
gates.substract = function (n) return n-3 end
gates.add ("substract", "operation", "after multiply")
```

And now `operation` returns `11` when fed 4. This was done by simply adding an optional third argument to the `add` operation; this specifies where the new gate(s) should be added in the l-gate: by default, it is the end of the list, but you can say `first` to put the gate(s) at the beginning or `before` *<name>* or `after` *<name>* to make the insertion before or after the gate called *<name>*; instead of a name, you can also use `first` or `last` to denote the first and last gates of the list, so here we could have used `before last` instead of `after multiply`.

Finally, you can also `remove` a gate:

```
gates.remove("substract", "operation")
```

and now `operation` is what it was before.

Handling arguments When a gate is called, it may take arguments ; in the case of an l-gate, those arguments are passed to each subgates, one after the other. Unlike gates in TEX, though, if a gate expects more arguments than passed to the l-gate it belongs too, no empty argument is added. For instance :

```
gates.mygate = function (a, b)
  print ("First: " .. tostring(a),
         "Second: " ..  tostring(b))
end
gates.list {"mylist"}
gates.add ("mygate", "mylist")
gates.mylist("one")
```

will print :

```
First: one          Second: nil
```

Also, there is by default no ‘common argument pool’, as there was in TEX : a gate (in an l-gate) receives what the previous one returns and nothing else ; in other words if l-gate L receives four arguments and subgate A returns only three, then subgate B will receive three arguments. Consequently, all gates should return properly if arguments are to be passed.

But that is only default behavior. Lua gates can be made to work like TEX gates (to some extent), by using `autoreturn`:

```
gates.autoreturn ("mygate", true)
gates.autoreturn ("mygate", "mylist", true)
```

When `autoreturn` is set for a gate (either globally, as in the first line, or locally, as in the second), missing arguments will be restored when the gate returns. To qualify as a missing argument, the following conditions should hold : first, the value is `nil` (i.e. either `nil` was explictly returned, or nothing was returned) ; second, no real argument follows. For instance, suppose `mygate` is defined as follows :

```
gates.mygate = function (A, B, C, D)
  return nil, X
end
```

If `autoreturn` is false, the following gate will receive `nil` and `X`; if it is true, the arguments will be restored to `nil`, `X`, `C` and `D`; the first argument isn't restored, even though it is `nil`, because it is followed by real argument `X`.

One can also uses `autoreturn` to completely disregard whatever a gate returns ; this happens when `autoreturn` is a function instead

of a boolean ; the function is passed the original arguments given to the gate, and what it returns overrides the gate's return values. For instance, given the same definition of `mygate`, if `autoreturn` is :

```
gates.autoreturn ("mygate",
  function (A, B, C, D) return D, C, B, A end)
```

then no matter what `mygate` returns (including what it returns on several iterations of itself, if any), the following gate will always receive the original arguments in reverse order. This can be particularly useful if the gate is controlled by an `iterator`, in which case the arguments it receives (e.g. the entries of a table) differ from the original ones (e.g. the table itself), yet you still want to pass the latter to the following gate.

L-gates automatically returns whatever their last gate returns. Also, unlike the TEX implementation (but in line with the Lua language), when a gate returns outside an l-gate, the returned values can be used, as with any other function.

4.3 Controlling gates

Here it is explained how gates can be controlled externally.

`Status` A gate's status can be set with the `open`, `ajar`, `skip` and `close` actions ; they take at least one argument, either a table :

```
gates.open ({"mygate", "myothergate"})
```

or a simple string if only one gate is to be affected :

```
gates.open ("mygate")
```

With one argument, the actions set the global status ; if a second argument is present, it is the l-gate where the local status is to be set :

```
gates.ajar ({"mygate", "myothergate"}, "mylist")
```

In this case, the first argument can denote a relative position of the form `before <gate>` or `after <gate>` :

```
gates.ajar ("before mygate", "mylist")
```

Conditional To make a gate depends on an external state of affair rather than status only, `conditional` can be used. The syntax is the same as for status, except a second (in case of global conditional) or third (in case of local conditional) argument is given. This should be a function, and the gate's execution depends on what the function returns : the gate is executed the function returns nothing or `false`. The arguments that are to be passed to the gate

are passed beforehand to the conditional function. For instance, if a gate receives arguments `A` and `B`, and its conditional function is `ControlFunction`, then the gate's execution can be schematized as:

```
if ControlFunction(A, B) then
  -- Execute gate
end
```

If you don't want a gate to depend on a conditional anymore, you can declare something like:

```
gates.conditional ("mygate", "mylist",
  function () return true end)
```

Loop The `loop` action allows a gate to be repeated; the syntax is the same as for `conditional`, and the loop will be repeated as long as the function evaluates to true (as with a `while` loop). Also, the gate's arguments are repeatedly passed to the loop conditional. To delete a gate's loop, use `loop` with `nil` as the third argument. As an example, the following will print number from `1` to `4`:

```
function gates.mygate (n)
  print(n)
```

```
  return n + 1
end
gates.loop("mygate", function (n) return n < 5 end)
gates.mygate(1)
```

Loop until The `loopuntil` action is like `loop`, except the gate is repeated until the conditional evaluates to false. Also, the conditional is evaluated after the gate is executed, so the execution takes place at least once. If both `loop` and `loopuntil` are set for a gate, the latter is ignored.

Iterator The `iterator` action is a bit more complex. If set, it is fed the arguments passed to the gate and should return a function, plus possibly a state, plus possibly an initial variable ; in other words, it should return whatever fits a `for` loop in Lua. Then the function is called with the state and variable, and the gate itself is called on whatever this function returns ; the process repeats until the function returns `nil` as its first argument. As an example :

```
gates.mygate = function (key, value)
  print("The value of " .. key .. " is " .. value)
end
gates.iterator ("mygate", pairs)
gates.mygate({x = 55, y = 22})
```

will print:

```
The value of x is 55
The value of y is 22
```

Here `pairs` was used to return the function, state and variable mentioned above; but of course you can make you own function:

```
gates.mygate = function (w)
  print(w .. " has " .. #w .. " characters.")
end
gates.iterator ("mygate",
  function (s)
    local t = string.explode(s)
    local i = 0
    local function enum ()
      i = i+1
      return t[i]
    end
    return enum
  end)
gates.mygate("two words")
```

And the result is:

```
two has 3 characters.
words has 5 characters.
```

Here the function registered in `iterator` produces a single function, without a state, and goes through all the entries of the table created by splitting the original string. In sum, what `iterator` expects is what Lua's generic `for` expects, and understanding the latter is understanding the former.

An important point to keep in mind when using `iterator` is that there is a discrepancy between the arguments passed to the gate and the ones it really processes ; in the first example, `mygate` is called with a table, but it receives two strings. This means that its definition doesn't match the way it is called ; in that respect, unlike `loop` and associates above, you just can't impose an `iterator` on a aready defined gate and expect everything to be fine, since its definition probably won't match the arguments it will now receive.

Another point is the return values, if any ; during the iteration, the gate's return values are ignored ; then the last ones are passed to the following gate, possibly augmented with some of the original arguments if `autoreturn` is `true`. Note that the original arguments here are those passed to the gate before the iteration states. This holds if `autoreturn` is a function too :

```
gates.mygate = function (what, ever)
```

```
  -- whatever.
end
gates.iterator ("mygate", pairs)
gates.autoreturn ("mygate",
  function (t) return t end)
gates.mygate (mytable)
```

Here `mygate` will loop on the entries in `mytable` (thanks to `pairs`), and once the iterations are over, `mytable` is returned.

Of course, `iterator` can be used with l-gates too ; that works the same : whatever the iterator returns is simply passed to the subgates. Finally, if a gate has either `loop` or `loopuntil`, `iterator` is ignored.

Gates can be created and manipulated by actions, as we've done up to now, but they can also be declared much faster. First, `def` and `list` take tables as their arguments ; that is for a good reason : entries indexed with certain keys are equivalent to actions. The keys are `autoreturn`, `status`, `conditional`, `loop`, `loopuntil` and `iterator` and setting them when declaring a gate is like globally setting the associated action (except there is a single action for `status`, which takes a string as its value : `open`, `ajar`, `skip` or `close`). Thus the following defines an m-gate with global status `ajar` and a `loop`:

4.4 The shorthand notation

```
gates.def {"mygate",
           status = "ajar",
           loop = function (n) return n < 5 end,
           function (n)
             print(n)
             return n + 1
           end}
```

Note that the entries can be given in any order as long as the gate's name is at index 1 and its definition at index 2.

L-gates have another property: as said above, the entry at index 1 should be the gate's name; but you can put tables representing gates at index 2, 3, etc., and they represent the l-gate's subgates. Those tables are the same as the ones passed to `def` and `list`; the only difference is that setting of `autoreturn`, `status`, etc., is local to the l-gate where they are added. The following code creates l-gate `mylist` with subgates `mygate` and `myothergate`, the former subject to a local loop:

```
gates.list {"mylist",
  {"mygate", loop = function (n) return n < 5 end,
    function (n)
     print (n)
     return n + 1
```

```
    end},
  {"myothergate",
    function (n)
     print("We're done!")
    end}}
```

Since the subtables are the same as the tables passed to `def` and `list`, it means that an l-gate thus declared can host subgates too :

```
gates.list {"mylist",
  {"mysublist",
      {"mygate", function () ... end},
      {"myothergate", function () ... end}},
  {"anothergate", function () ... end}}
```

If you want to add an already existing gate, you can do so either by using a simple string instead of a table, or (if you want to set `status` and associates) a table with the name at index 1 and nothing at higher indices :

```
gates.list {"mylist",
  "AnExistingSubgate",
  {"AnExistingSubgateWithOption", status = "ajar"}}
```

Note however that the second version is similar in form to an l-gate defined without subgates ; this means that you can't redefine an l-gate thus, but the situations where you would like to do so (redefining an existing l-gate, and redefining it without subgates, and in shorthand notation) aren't many. In that case you should use `list` and `add` instead. (On the other hand, if you do specify subgates, redefinition will happen with an l-gate with that name already exists ; in other words, it won't be interpreted as inserting an existing l-gate and add subgates to it.)

4.5 Gate families

The `new` action takes a string (the family name) as its sole argument an return a calling table :

```
MyGates = gates.new ("MyFamily")
```

`MyGates` can now be used like `gates` (to which is associated the `gates` family), except `MyFamily` will be used as the family name when necessary.

A table's family is stored in the `family` entry ; note that it is not an action, hence not a function, but a string :

```
print(MyGates.family)
```

Both `type` and `status` return a number: `type` returns 0 if there is no gate with that name, 1 if it is an m-gate and 2 for an l-gate; `status` returns 0 for a non-existing gate, and 1 to 4 for `open`, `ajar`, `skip` and `close` respectively; `status` can also be used with one or two arguments, indicating global or local status respectively:

```
gates.status ("mygate")
gates.status ("mygate", "mylist")
```

The `subgates` action passes the names of all the subgates in a given l-gate to a function. For instance, the following prints all `mylist`'s subgates and their types:

```
gates.subgates("mylist",
function (g)
  local t
  if gates.type(g) == 1 then
    t = "(m-gate)"
  else
    t = "(l-gate)"
  end
  print (g, t)
end)
```

The `show` action can be used to display a gate's construction and conditions. It is displayed as the shorthand notation. The `trace` action takes a number and works as follows: if the number is 0, gates of the associated families aren't traced; if it is 1, it is mentionned if they are called (in an l-gate) and executed or not, and why; with 2, it works like 1, except the arguments passed to the gates are shown too.

Reference manual for Lua

There are some differences between gates in TEX and gates in Lua, owing to the differences in syntax between the two languages. That said, the same operations can be found in both interfaces.

In what follows, *<gate list>* denotes either a table with gate names at successive indices (e.g. `{'mygate', 'myothergate'}`) or a single string denoting a single gate.; *<gate spec>* denotes either a *<gate list>* or a relative position of the form `before` <gate> or `after` <gate> (the latter case only if an l-gate is also specified, obviously).

`add (`*<gate list>*`, `*<l-gate>*`[, `*<position>*`])`

Adds the gates in *<gate list>* to *<l-gate>*. The optional *<position>* should be one of the following: `first`, meaning the gates will be added at the beginning of the l-gate; `last`, meaning they will be added at the end (so this is similar to no *<position>*); `before` <name>, and the addition will occur before gate *<name>* in the l-gate

(it should of course exist); `after <name>`, and the addition will occur after gate *<name>*. Note that *<name>* can be `first` or `last`, denoting the first and last gates in *<1-gate>* (`before first` and `after last` are obviously synonymous with `first` and `last`).

`ajar (<gate spec>[, <1-gate>])`

Without *<1-gate>*, sets the global status for the gates in *<gate spec>* to `ajar`. With *<1-gate>*, sets the local status in *<1-gate>* to `ajar` for the gates in *<gate spec>*.

`autoreturn (<gate spec>[, <1-gate>], <boolean or function>)`

Without *<1-gate>*, sets the global `autoreturn` for the gates in *<gate spec>*; with *<1-gate>*, the local `autoreturn` is done. When a gate's `autoreturn` is set to `true`, arguments will be restored if it returns less than what it was passed; if `autoreturn` is a function, it is passed the original arguments passed to the gates and what it returns will be the ultimate return values of the gate. In the latter case, the gate's conditions are ignored, i.e. the arguments passed to `autoreturn` are the ones fed to the gate before it loops or iterates (if it does that).

`close (<gate spec>[, <1-gate>])`

Without *<1-gate>*, sets the global status for the gates in *<gate spec>* to `close`. With *<1-gate>*, sets the local status in *<1-gate>* to `close` for the gates in *<gate spec>*.

`conditional (<gate spec>[, <1-gate>], <function>)`

Without *<1-gate>*, sets the global conditional for the gates in *<gate*

spec> to <*function*>. With <*l-gate*>, sets the local conditional in <*l-gate*> to <*function*> for the gates in <*gate list*>. That the conditional is a function means that the gate(s) will be executed only if <*function*> returns anything but `nil`.

`copy (`<*gate1*>`, `<*gate2*>`)`

Defines <*gate1*> as <*gate2*> (either an m- or l-gate). The current global status is also copied. If <*gate2*> is an l-gate, its gate list is also copied, along with the current status of the gates it contains. On the other hand, if <*gate2*> occurs in some l-gate(s), this information isn't copied.

`def (`<*table*>`)`

Defines an m-gate whose name is the entry at index `1` in <*table*>, and the defintion the function at index `2`; `autoreturn`, `status`, `conditional`, `loop` and `loopuntil` can also be used as key to globally specify those settings.

Assigning to an entry in the general `gates` table is a shorthand for `def`; in other words, the following two lines are synonymous, provided there isn't a gate action called `foo`.

```
gates.def {"foo", function (...) ... end}
gates.foo = function (...) ... end
```

In the second case, if what is assigned isn't a function, then a new entry is simply added to the `gates` table, as if it were a simple

table, unless the index is an existing action or gate (in which case an error is raised). Note that this entry will then be unavailable to host a gate if redefined.

`execute (<gate>[, ...])`
: Calls *<gate>* with the other arguments. There exists two shorthands : `gates.<gate>(...)`, provided *<gate>* doesn't clash with an action's name, and `gates.execute.<gate>(...)`. Those shorthands are also the only way to retrieve the gate itself instead of executing it.

`family`
: The family associated with a table created with `new`. This is a string, not a function.

`iterator (<gate spec>[, <1-gate>], <function>)`
: Without *<1-gate>*, sets the global iterator for the gates in *<gate spec>* to *<function>*. With *<1-gate>*, sets the local iterator in *<1-gate>* to *<function>* for the gates in *<gate list>*. The *<function>* is passed the gates' original arguments and should return a function, plus possibly a state and also a variable ; then the function will be called repeatedly with the state and variable and the gates will be called on what the function returns until it returns `nil` as the first argument. What the gates return is ignored until the last iteration, in which case it is passed to the next gate (unless `autoreturn` is set). To delete a gate's iterator, use the same action with `nil` as *<function>* (but that's generally not a good idea, since

gates with iterators are tailored to the arguments the iterator returns, not to the arguments they are called with).

`list (`*`<table>`*`)`

Declares an l-gate whose name is the entry at index `1` in *`<table>`*. Additional entries can be specified as with `def`. Tables at indices `2`, `3`, etc., are subgates created and added at once to *`<table>`*, with `status` and associates indicating local (not global) settings. Existing gates can also be added without being redefined, by giving their names (a simple string) instead of a full table, or a table with nothing at indices `2` and higher.

`loop (`*`<gate spec>`*`[, `*`<l-gate>`*`], `*`<function>`*`)`

Without *`<l-gate>`*, sets the global while-loop for the gates in *`<gate spec>`* to *`<function>`*. With *`<l-gate>`*, sets the local while-loop in *`<l-gate>`* to *`<function>`* for the gates in *`<gate list>`*. The gates will be executed again as long as *`<function>`* evaluates to true. To delete a gate's while-loop, use the same action with `nil` as *`<function>`*.

`loopuntil (`*`<gate spec>`*`[, `*`<l-gate>`*`], `*`<function>`*`)`

Without *`<l-gate>`*, sets the global until-loop for the gates in *`<gate spec>`* to *`<function>`*. With *`<l-gate>`*, sets the local until-loop in *`<l-gate>`* to *`<function>`* for the gates in *`<gate list>`*. The gates will be executed again as long as *`<function>`* doesn't evaluate to true. To delete a gate's until-loop, use the same action with `nil` as *`<function>`*.

`open (<gate spec>[, <l-gate>])`

Without *<l-gate>*, sets the global status for the gates in *<gate spec>* to open. With *<l-gate>*, sets the local status in *<l-gate>* to open for the gates in *<gate spec>*.

`remove (<gate list>, <l-gate>)`

Removes the gates in *<gate list>* from *<l-gate>*.

`show (<gate>)`

Writes to the terminal (and log file, if in LuaTEX) *<gate>*'s type, its number of arguments, global status, conditional, loops (if specified), `autoreturn` if set to true, and recursively its subgates if *<gate>* is an l-gate (showing the local settings).

`skip (<gate spec>[, <l-gate>])`

Without *<l-gate>*, sets the global status for the gates in *<gate spec>* to `skip`. With *<l-gate>*, sets the local status in *<l-gate>* to `skip` for the gates in *<gate spec>*.

`status (<gate>[, <l-gate])`

Without *<l-gate>*, returns the global status of *<gate>*; with *<l-gate>*, the local status is returned: 1 if the gate is open, 2 if it is ajar, 3 if it is to be skipped, and 4 if it is closed; if there *<gate>* doesn't exist, of doesn't exist in *<l-gate>*, 0 is returned.

`subgates (<l-gate>, <function>)`

Executes *<function>* with each gate (represented by its name as a string) in *<l-gate>*.

`trace (<number>)`

If *<number>* is 0, gate operations aren't reported; if 1, encountered gate are reported, along with their status and conditional when necessary; if 2, arguments passed to gates that are executed are also displayed. Tracing affects only gates of the family associated with the calling command.

`type (<name>)`

Returns the type of *<name>*: 0 if it isn't a gate, 1 if it is an m-gate, 2 if it is an l-gate.