CafeOBJ Commands Quick Reference

(for interpreter version 1.4.8)

Notation

Keywords apper in `type setter face`, when presented in the form like 'x(yz)' it means the keyword 'xyz' can be abbreviated to 'x'. '[something]' means 'something' is optional. `|` is used for listing alternatives. Slanted face, e.g., *variety* is used when it varies (a meta-variable) or is an expression of some language. For example, *modexp* is for module expressions and *term* is for terms (you should know what these are); others should easily be understood by their *name*s and/or from the context.

Starting CafeOBJ interpreter

To enter CafeOBJ, just type its name: `cafeobj`

'`cafeobj -help`' will show you a summary of command options.

Leaving CafeOBJ

`q(uit)` exits CafeOBJ.

Getting Little Help

Typing `?` at the top-level prompt will print out a list of whole top-level commands.

Escape

There would be a situation that you hit `return` expecting some feedback from the interpreter, but it does not respond. This occurs when the interpreter expects some more inputs from you thinking preceding input is not yet syntactically complete. If you encounter this situation, and you don't know what the interpreter expects, simply type in `esc`(escape key) and `return`, then it will immediately be back to you discarding preceding input and makes a fresh start.

Rescue

Occasionally you may meet a strange prompt `CHAOS>>` after some error messages. This happens when the interpreter caused some internal errors and could not recover from it. Try typing `:q`, this may resume the session if you are lucky.

Sending interrupt signal (typing `C-c` from keyboard, or if you are in Emacs, some key sequence specific to the *mode* you are in) forces the interpreter to break into underlying Lisp, and you will see the same prompt as the above. This might be useful when you feel the interpreter get confused. `:q` also works for returning to CafeOBJ interpreter from Lisp.

Setting Switches

Switches are for controlling the interpreter's behaviour in several manner. The general form of setting top-level switch is:

`set` *switch value*

In the following, the default value of a switch is shown underlined.

switch	**value**	**what?**
***		– switches for rewriting
`trace whole`	`on`|<u>`off`</u>	trace top-level rewrite step
`trace`	`on`|<u>`off`</u>	trace every rewrite step
`step`	`on`|<u>`off`</u>	stepwise rewriting process
`memo`	<u>`on`</u>|`off`	enable term memoization
`clean memo`	`on`|<u>`off`</u>	clean up term memo table before normalization
`stats`	<u>`on`</u>|`off`	show statistics data after reduction
`rwt limit`	*number*	maximum number of rewriting
`stop pattern`	[*term*] .	stop rewriting when meets
`mel sort`	`on`|<u>`off`</u>	compute result sort with sort membership predicates
`reduce conditions`	`on`|<u>`off`</u>	reduce conditional part in apply command
`verbose`	`on`|<u>`off`</u>	set verbose mode
`exec trace`	`on`|<u>`off`</u>	trace concurrent execution
`exec limit`	*number*	limit maximum number of concurrent execution
`exec normalize`	<u>`on`</u>|`off`	reduce term before and after each transition
`exec all`	<u>`on`</u>|`off`	find all solutions of `=(*)=>`
***		– switches for system's behaviour
`include BOOL`	<u>`on`</u>|`off`	import BOOL implicitly
`incude RWL`	<u>`on`</u>|`off`	import RWL implicitly
`include FOPL-CLAUSE`	<u>`on`</u>|`off`	import FOPL-CLAUSE implicitly
`auto context`	`on`|<u>`off`</u>	change current context in automatic
`auto reconstruct`	`on`|<u>`off`</u>	perform automatic reconstruction of modules if it is inconsistent
`reg signature`	`on`|<u>`off`</u>	regularize module signature in automatic
`check regularity`	`on`|<u>`off`</u>	perform regularity check of signature in automatic
`check compatibility`	`on`|<u>`off`</u>	perform compatibility check of TRS in automatic
`check builtin`	<u>`on`</u>|`off`	perform operator overloading check with built-in sorts
`select term`	`on`|<u>`off`</u>	system selects a term from ambiguously parsed terms
`quiet`	`on`|<u>`off`</u>	system mostly says nothing
		– show/display options
`all axioms`	`on`|<u>`off`</u>	print all axioms in "`sh(ow)` *modexp*" command
`show mode`	<u>`:cafeobj`</u> | `:chaos`	set syntax of printed modules or views
`show var sorts`	`on`|<u>`off`</u>	print variables with sorts
`print mode`	<u>`:normal`</u> | `:fancy` | `:tree` | `:s-expr`	set term priting form
***		– miscellaneous settings
`libpath`	*pathname*	set file search path
`print depth`	*number*	maximum depth of terms to be printed
`accept =*= proof`	`on`|<u>`off`</u>	accept system's automatic proof of congruency of `=*=`

The default value of *pathname* of `set libpath` command is '$cafeob-

jhome/lib, $cafeobjhome/exs', where '$cafeobjhome' varies depending on the installation options of your interpreter. Normally, it is `/usr/local/lib/cafeobj1.4`.

The default value of *number* in 'set rwt limit' command is 0 meaning no limit counter of rewriting is specified.

Omitting *term* in `set stop pattern` sets the stop pattern to empty, i.e., no term will match to the pattern.

Examining Values of Switches

`show switch`	print list of available switches with their values
`show switch` *switch*	print out the value of the specified *switch*

Setting Context

`select` *modexp*

This sets the context of the interpreter (**current module**) to the module specified by *modexp*. It must be written in single line. When you type in *modexp*, the '`;`<newline>' treated as a line continuation (that is, it is effectively ignored), so that you can type in multiple lines for long module expressions. Note that one or more blank characters are required before `;`.

Inspecting Module

`sh(ow)` and `desc(ribe)` commands print information on a module. In the sequel, we use a meta-variable *show* which stands for either `sh(ow)` or `desc(ribe)`. Most of the cases, giving `desc(ribe)` for *show* gives you more detailed information.

show *modexp*	prints a module *modexp*. giving '`.`" as *modexp* shows the current module
show `sorts` [*modexp*]	prints sorts of *modexp*
show `ops` [*modexp*]	prints operators of *modexp*
show `vars` [*modexp*]	prints variables of *modexp*
show `params` [*modexp*]	prints parameters of *modexp*
show `subs` [*modexp*]	prints direct submodules of *modexp*
show `sign` [*modexp*]	prints `sorts` and `ops` combined

modexp must be given in an one line. The same convention for long module expressions is used as that of `select` command (see **Setting Context** above.) If the optional [*modexp*] is omitted, it defaults to the current module. Optionally supplying `all` before `sorts`, `ops`, `axioms`, and `sign`, i.e., `desc all ops` for an instance) makes printed out information also include imported sorts, operators, etc. otherwise it only prints own constructs of the *modexp*.

The following *show* commands assume the current module is set to some module.

show `sort` *sort*	prints information on sort *sort*
show `op` *operator*	prints information on operator *operaotr*

For inspecting submodules or parameters, the following *show* commands are useful:

show `param` *argname*	prints information on the parameter
show `sub` *n*	prints information on the *n*th direct submodule

argname can be given by position, not by name.

You can see the hierarchy of a module or a sort by the follwing `sh(ow)` commands:

`sh(ow) module tree` *modexp*	prints pictorial hierarchy of module. specifying `.` as *modexp* shows the hierarcy of the current module
`sh(ow) sort tree` *sort*	prints hierarchy of sort pictorially

Evaluating Terms

`red(uce)` [`in` *modexp* `:`] *term* `.`

`exec(ute)` [`in` *modexp* `:`] *term* `.`

`reduce` reduces a given term *term* in the term rewriting system derived from *modexp*. `execute` is similar to `reduce`, but it also considers axioms given by **transition** declarations. In both cases, omitted '`in` *modexp* `:`' defaults to the current module.

The result term of `reduce` and `execute` is bould to special variables `$$term` and `$$subterm` (see the next section).

Let Variables and Special Variables

`let` *let-variable* `=` *term* `.`

let-variable is an indentifier. Assuming the current module is set, `let` binds *let-variable* to the given term *term*. Once set, *let-variable* can be used wherever *term* can apper.

You can see the list of let bidings by:

`sh(ow) let`

There are two built-in special variables in the system:

`$$term`	bound to the result term of `reduce`, `execute`, `parse`, or `start` commands.
`$$subterm`	bound to the result of `choose` command

Let variales and special variables belongs to a context, i.e., each context has its own let variables and special variables.

Inspecting Terms

`parse` [`in` *modexp* `:`] *term* `.`

`parse` parses given term *term* in the module *modexp* (if omitted, parses in the current module) and prints the result. The result is bound to special variables `$$term` and `$$subterm`.

The following `sh(ow)` command assumes the current module, and prints the term.

`sh(ow) term` [*let-variable*] [`tree`]

let-variable can be a name of *let-variable*, `$$term` or `$$subterm`, if omitted the term bound to `$$term` is printed. If optional `tree` is supplied, it prints the term tree structure.

Opening/Closing Module

`open` *modexp*	opens module *modexp*
`close`	close the currently opening module

Opening module can be modified, i.e., you can declare new sorts, operators, axioms. You can open only one module at a time.

Applying Rewrite Rules

Start The initial target (entire term) is set by `start` command.

`start` *term* .

This binds two special variables `$$term` and `$$subterm` to *term*.

Apply `apply` command applies actions to (subterm of) `$$term`.

`apply` *action range selection*

You specify an action by *action*, and it will be applied to the target (sub)term specified by *selection*.

range is either `within` or `at`: `within` means at or inside the (sub)term specified by the *selection*, and `at` means exactly at the *selection*.

Action *action* can be the followings:

`red(uction)`	reduce the selected term
`exec`	execute the selected term
`print`	print the selected term
rule-spec	apply specified rule to the selected term

Rule-Spec *rule-spec* specifies the rule with possibly substitutions being applied, and given by

[`+ | -`][*modexp*].*rule-name* [*substitutions*]

The first optional '`+ | -`' specifies the direction of the rule; left to right(if `+` or omitted) or right to left (if `-`).

A rule itself is specified by '[*modexp*].*rule-name*]'. This means the rule with name *rule-name* of the module *modexp* (if omitted, the current module). *rule-name* is either a label of a rule or a number which shown by `sh(ow) rules` command (see **Showing Available Rules below**.)

substitution binds variables that apper in the selected rule before applying it. This has the form

`with` *variable* `=` *term* `,` ...

Showing Available Rules To see the list of the rewrite rules, use

`sh(ow)` [`all`] `rules`

The list of the (all, i.e., includes imported rules if the optional `all` is supplied) available rules are printed with each of which being numbered. The number can be used for *rule-name* (see above).

Selection *selection* is a sequence of *selector* separated by keyword `of` specifying (sub)term of `$$term`:

selector { `of` *selector* } ···

selector	**description**
`term`	the entire term (`$$term`)
`top`	ditto
`subterm`	selects `$$subterm`
(*number* ···)	selects by position
[*number* `..` *number*]	by range in flattened term structure
{ *number* `,` ··· }	subset in flattened term structure

Step by Step Subterm Selection `choose` command selects a subterm of `$$subterm` and reset the `$$subterm` to the selected one.

`choose` *selector*

Matching Terms

`match` *term_spec* `to` *pattern*

term_spec specifies the term to be matched with *pattern*:

term_spec	**description**
`term`	`$$term`
`top`	ditto
`subterm`	`$$term`
`it`	ditto
term	ordinal term

pattern	**description**
`[ all ][+ | -] rules`	match with available rewrite rules
term	match with specified term

Stepper

If the switch `step` is set to `on`, invoking `reduce` or `execute` command runs into the term rewriting stepper. The stepper has its own command interpreter loop, where the following stepper commands are avilable:

`?`	print out available commands.
`n(ext)`	go one step
`g(o)` *number*	go *number* step
`c(ontinue)`	continue rewriting without stepping
`q(uit)`	leave stepper continuing rewrite
`a(bort)`	abort rewriting
`r(rule)`	prints current rewrite rule
`s(ubst)`	prints substitution
`l(imit)`	prints rewrite limit counter
`p(attern)`	prints stop pattern
`stop` [*term*]	set (unset) stop pattern
`rwt` [*number*]	set (unset) rwrite limit counter

You can also use families of `sh(ow)`(`desc(ribe)`) and `set` commands in stepper.

Reading In Files

`input` *file*	read in CafeOBJ program from *file*
`provide` *feature*	provide the *feature*
`require` *feature* [*file*]	require *feature*

Save and Restore

`save` *file*	save definitions of modules and views to *file*
`restore` *file*	restore definitions of modules and views
`reset`	recover definitions of built-in modules
`full-reset`	reset system to initial status
`save-system` *file*	save image of the interpreter to *file*

Protecting Your Modules

`protect` *modexp*	prevent the module from redefinition
`unprotect` *modexp*	allow moudle to be redefined

Little Semantic Tools

`check reg(ularity)` [*modexp*]	reports the result of regularity check of module
`check comat(ibility)` [*modexp*	reports the result of compatibility check of the module

For both commands, omitted *modexp* will perform the check in the current module.

The following `check` command assumes the current module:

`check laziness` [*operator*]

This checks strictness of *operator*. If *operator* is omitted all of the operators declared in the current modules are checked.

TRAM Compiler Interface

`tram compile` [*modexp*]

This compiles module *modexp* to Term Rewriting Abstract Machine. If *modexp* is omitted, it defaults to the current module. *modexp* must be given in an line. You can supply multiple lines by using '`;`<new-line>'.

To evaluate term in compiled module, use the following:

`tram exec` [`in` *modexp* :] *term*

Omitting '`in` *modexp* :' means the evaluation is performed in the current module. If the module *modexp* is not yet compiled, this compiles it implicitly, then perform the evaluation.

Miscellany

`ls` *pathname*	list contents of directories
`cd` *pathname*	change working directory of the interpreter
`pwd`	prints working directory
`!` *command*	fork shell *command*
`ev` *lisp*	evaluate lisp expression *lisp* printing the result
`evq` *lisp*	evaluate lisp expression *lisp*