Interactive CAL Environment (ICE) Manual

Contributors: Jennifer Chen, James Wright
Last modified: Nov 15th, 2007

CONTENTS

1 OVERVIEW 1

2 GETTING STARTED 1

2.1 Running ICE 1

2.2 ICE Settings 1

2.2.1 General Settings 1

Workspace 1

Module 1

Machine type 1

Command history size 2

# of runs for performance tests 2

Suppression of output 2

Suppression of incremental machine stats 2

2.2.2 Machine Settings 2

`org.openquark.cal.show_adjunct` 2

`org.openquark.cal.machine.debug_capable` 2

`org.openquark.cal.optimizer_level` 2

`org.openquark.cal.machine.lecc.source` 2

`org.openquark.cal.machine.lecc.runtime_statistics` 3

`org.openquark.cal.machine.lecc.runtime_call_counts` 3

`org.openquark.cal.machine.lecc.output_directory` 3

`org.openquark.cal.machine.lecc.code_generation_stats [module name]` 3

`org.openquark.cal.machine.lecc.debug_info` 3

`org.openquark.cal.machine.lecc.static_runtime` 3

`org.openquark.cal.machine.lecc.non_interruptible` 3

`org.openquark.cal.machine.lecc.strict_foreign_entity_loading` 3

`org.openquark.cal.machine.lecc.concurrent_runtime` 3

`org.openquark.cal.machine.g.runtime_statistics` 3

`org.openquark.cal.machine.g.call_counts` 3

2.3 Loading CAL modules 4

3 TUTORIALS 6

3.1 Using :find 6

3.2 Single-threaded Debugging 7

3.3 Multi-threaded Debugging 8

4 COMMAND REFERENCE 11

4.1 General 11

:q[uit] 11

:v[ersion] 11

:rs [module name] 11

:rsm [module name] 11

:rc 11

:rca 11
:rl........ 11
:sm <target_module_name> 12
:rr 12
:script <script file name> 12
:of <output file name> 12
:h[elp] [<topic>]........ 12

4.2 Info........ 13
:smd <dependee_module_name>........ 13
:si <dependent_module_names>........ 13
:t <expression>........ 13
:sps [module names] 13
:ss........ 13
:sts........ 13
:swi........ 13
:lcfs 13
:cfs 14
:lcafs [<module_name>] [<includeDependees>]........ 14

4.3 Find........ 14
:f[ind] [all] <entity_name> 14
:f[ind] ref <gem_name>........ 14
:f[ind] defn <gem_or_type_class_name> 14
:f[ind] instByType <type_name>........ 14
:f[ind] instByClass <type_class_name> 14
:f[ind] constructions <type_or_constructor_name>........ 14

4.4 Workspace 15
:lmw <dependent_module_names> 15
:adm <module_name> 15
:ldw <workspace_file_name>........ 15
:lw <workspace_file_name>........ 15
:lm........ 15
:sdw........ 15
:sdm 15
:rm <target_module_name>........ 15

4.5 Debug........ 15
:step........ 16
:step <CAL expression> 16
:re........ 16
:ret........ 16
:sus[pend]........ 16
:threads........ 16
:thread <thread id>........ 16
:te 16
:show........ 16
:show function 16
:show argnames........ 16
:show argTypes........ 17
:show <argument name>........ 17
:show stack [expanded]........ 17
:bp <qualified name> 17
:bp show........ 17
:bp clear 17
:trace on 17
:trace off........ 17

:trace <qualified name> 17
:trace clear 17
:trace arguments on 17
:trace arguments off 17
:trace threadname on 18
:trace threadname off 18
:fa <regExpr> 18
:fr <index> 18
:fs 18
:stt 18
:d <function_name> 18
:da 18
:ded 18

4.6 Benchmark 18
:pt <cal code> 18
:ptm <cal code> 19
:brf [benchmark file name] 19
:brl <benchmark label> 19
:sptr <number of runs> 19

4.7 CALDoc 19
:docgen <options> 19
:docm <module_name> 20
:docf <function_or_class_method_name> 20
:doct <type_name> 20
:docd <data_constructor_name> 20
:docc <type_class_name> 20
:doci <type_class_name> <instance_type_constructor> 20
:docim <method_name> <type_class_name> <instance_type_constructor> 20

4.8 Refactor 21
:rf imports [<module_name>] [nogroup] 21
:rf typeDecls [<module_name>] 21
:pp [<module_name>] [<function_name> [<function_name> ...]] [startLine endLine] 21
:ren[ame] [<category>] <old_identifier> <new_identifier> 21

4.9 Utilities 21
:jar [-verbose] (-main <functionName> <mainClassName> | -lib <moduleName> <libClassScope> <libClassName>)+ <outputJarName> [-src <outputSrcZipName>] 21
:car [-keepsource] [-nocws | -nosuffix] [-s] [-jar] [-d] <output directory> 22
:wsi [-l] <dependent_module_names> "<workspace_file_name>" 22
:gen[erate] deriving [compact] <type_name> 22
:gen[erate] deriving efficient <type_name> 22
:javaBinding <module name pattern> <target package> [internal] [test] 22

4.10 History 23
:spc 23
:pc <command number> 23
:npc <number of commands> 23

4.11 Other 23
:caf <CAF name> 23
:mt <machine_type> 23
:sms 23
:so 23

1 Overview

The Interactive CAL Environment (ICE) is a text-based shell that enables the evaluation of CAL expressions. It allows the user to load CAL modules, and then type in expressions to be evaluated in the context of those modules. ICE contains many built-in tools, such as tools for debugging, CALDoc generation and benchmarking.

2 Getting Started

2.1 *Running ICE*

To launch ICE, run the `ICE.bat` batch file on Windows, `ICE.command` file on Mac OS or the `ICE.sh` shell script on UNIX/Linux platforms. Another method to launch ICE is using the source code included in the Open Quark distribution. Refer to the document `Using CAL in Eclipse` for instructions on how to install the source files in Eclipse.

2.2 *ICE Settings*

ICE settings are displayed during start up. Any time during the ICE session, use `:ss` command to display the current settings. To view the descriptions about various settings, use `:h[elp] options` command.

2.2.1 General Settings

These settings can be adjusted during an ICE session.

Workspace

Specifies the workspace definition currently in use. To set the workspace definition from start up, define either the property `org.openquark.cal.ice.workspace` using fully qualified path name or `org.openquark.cal.ice.workspace.filename` using an unqualified filename. For more information see Workspace commands.

Module

Specifies the working module. To specify the initial working module, define the property `org.openquark.cal.ice.module` as VM argument.

Machine type

By default, the machine type is LECC, a high-performance machine which compiles to Java bytecodes. An interpreter (G-machine) is available as well. The machine type to use upon start up can be defined by setting the property `org.openquark.cal.machineType` as VM argument. Subsequent machine type changes can be set using the `:mt` command.

Command history size

The command history size represents the number of recent commands which are remembered for later use when performing history commands. The default is 10 commands but the size can be changed. Note that the size change is specific to the current ICE session. For more information, see History commands.

# of runs for performance tests

The number of runs to carry out for performance tests. The default is 10 runs but can be changed. For more information, see Benchmark commands.

Suppression of output

Flag to stop the display of outputs from CAL expressions. It is by default off. Use the `:so` command to toggle the flag.

Suppression of incremental machine stats

Flag to suppress the display of incremental machine stats. It is by default on. Use the `:sms` command to toggle the flag.

2.2.2 Machine Settings

Define the environment properties by setting –D VM arguments. These settings cannot be changed during an ICE session.

General machine properties

`org.openquark.cal.show_adjunct`

If this is defined the generated I/O code for the executing expression will be displayed.

`org.openquark.cal.machine.debug_capable`

Defining this activates the debugging ability within ICE. It enables a trace statement to be put at the beginning of each generated function, as well as the ability to set breakpoints on CAL functions and examine the program state when suspended.

`org.openquark.cal.optimizer_level`

If this is defined as level 1, the CAL Global Optimizer is enabled. To disable the optimizer set the level to 0. By default the optimizer is off.

LECC machine specific properties

`org.openquark.cal.machine.lecc.source`

If this is defined the LECC machine will generate Java source code, otherwise Java byte-code is directly generated.

`org.openquark.cal.machine.lecc.runtime_statistics`

Defining this generates runtime statistics for the LECC machine such as reduction count and function count.

`org.openquark.cal.machine.lecc.runtime_call_counts`

Defining this generates runtime statistics for function call frequency.

`org.openquark.cal.machine.lecc.output_directory`

The value of this variable determines the destination of generated LECC files

`org.openquark.cal.machine.lecc.code_generation_stats [module name]`

Defining this displays information about generated code for the module specified, or for all modules if none specified. Information includes number of optimized applications and number of switches.

`org.openquark.cal.machine.lecc.debug_info`

Defining this will cause a stack trace to be dumped to the console when `Prelude.error` is called.

`org.openquark.cal.machine.lecc.static_runtime`

If this is defined and bytecode generation enabled, the LECC machine will have runtime class files generated to and loaded from disk.

`org.openquark.cal.machine.lecc.non_interruptible`

If this is defined, the LECC runtime will not check whether a quit has been requested.

`org.openquark.cal.machine.lecc.strict_foreign_entity_loading`

If this is defined, foreign entities corresponding to foreign types and foreign functions will be loaded eagerly during deserialization.

`org.openquark.cal.machine.lecc.concurrent_runtime`

If this is defined, LECC run-time supports concurrent reduction of CAL programs on a single execution context.

G-Machine specific properties

`org.openquark.cal.machine.g.runtime_statistics`

Defining this generates runtime statistics for the G-machine such as instruction count.

`org.openquark.cal.machine.g.call_counts`

Defining this generates runtime statistics for function call frequency.

2.3 Loading CAL modules

Only expressions can be entered at the ICE prompt. Function definitions, type declarations, and other declarations must appear in modules. The best way to enter these declarations is to create your own module file. The declarations can be typed into the module file, which is then reloaded into the ICE environment.

To load a CAL module, two key steps are required. First we need to create and save the file containing declarations with .cal file extension. Second, the CAL workspace (.cws) file for ICE needs to be updated to include the new module. An example for workspace declaration is "`cal.samples.cws`". The current workspace declaration file to your ICE session can be found using the `:ss` command.

Here are the steps that could be used in an ICE session to try some example code using the binary distribution. (Instruction for source distribution will be given below)

1. Use a text editor to create a file named `UserGuideExamples.cal`.
2. Copy and paste the following code into the new file:

```
module UserGuideExamples;
import Cal.Core.Prelude using
    typeConstructor = Int, Double, String, Boolean, Char,
                      Integer, JObject, JList, Maybe, Ordering;
    dataConstructor = False, True, LT, EQ, GT, Nothing, Just;
    typeClass = Enum, Eq, Ord, Num, Inputable, Outputable;
    function =
        add, append, compare,
        concat, const, doubleToString, equals,
        error, fromJust, fst, input, intToString, isNothing,
        isEmpty, max, mod, not, output, round, seq, snd,
        toDouble, field1, field2, field3, upFrom, upFromTo;
    ;
import Cal.Collections.List using
    function =
        all, chop, filter, foldLeft, foldLeftStrict, foldRight,
        head, intersperse, last, list2, map, product, reverse,
        subscript, sum, tail, take, zip, zip3, zipWith;
    ;
import Cal.Collections.Array;
import Cal.Core.Bits;
import Cal.Core.Debug;
import Cal.Core.Dynamic using
    typeConstructor = Dynamic;
    function = fromDynamic, fromDynamicWithDefault, toDynamic;
    ;
import Cal.Utilities.Math using
    function = truncate;
    ;
import Cal.Utilities.StringNoCase;
import Cal.Core.String;
```

3. Save `UserGuideExamples.cal` to "`Quark\bin\cal\debug\CAL`" folder.

4. Use a text editor to add the following line to the end of the workspace declaration file "`cal.samples.cws`" found in the "`samples\simple\Workspace Declaration`" folder.

```
StandardVault UserGuideExamples
```

5. Run `ICE.bat` (or `ICE.sh`) in the unzipped Quark folder.
6. Set the current module to be `UserGuideExamples` by entering the following at the ICE prompt:

```
:sm UserGuideExamples
```

7. In a text editor, add some function definitions to the end of `UserGuideExamples.cal` and save the file. For example:

```
myFactorial :: Integer -> Integer;
myFactorial n =
    product (upFromTo 1 n);
```

8. Use the :`rc` command from the ICE prompt to recompile and load all changed module files:

```
:rc
```

9. Enter some expressions at the ICE prompt for evaluation:

```
myFactorial 5
```

If you are using the source distribution within Eclipse to run ICE, follow the steps above except for the following:
In step 3, save `UserGuideExamples.cal` to "`\Quark\src\CAL_Platform\src\CAL`"
In step 4, the default workspace will be "`ice.default.cws`". Add the declaration to the file found at "`Quark\src\CAL_Platform\src\Workspace Declarations\ice.default.cws`".
In step 5, run ICE according to the instruction in the document `Using CAL in Eclipse`.

3 Tutorials

3.1 *Using :find*

In section 2.3, a sample module named `UserGuideExamples.cal` was generated. Add the following code to the end of the file then use the command `:rc` to recompile.

```
myFindExample :: Boolean -> Boolean;
myFindExample b = b;

myFindRefExample :: Boolean;
myFindRefExample = myFindExample True;

data public MyFindDataExample =
    public MyFind
    deriving Eq, Ord;
```

After the module has been recompiled, type `:f myFindExample` on the command line which will generate 3 search results within the workspace:

```
UserGuideExamples: (line 37, column 1) - UserGuideExamples.myFindExample
- myFindExample :: Boolean -> Boolean;

UserGuideExamples: (line 38, column 1) - UserGuideExamples.myFindExample
- myFindExample b = b;

UserGuideExamples: (line 41, column 20) - UserGuideExamples.myFindExample
- myFindRefExample = myFindExample True;
```

We can refine the search and use `:f ref myFindExmple` command to only generate the third result above (on line 41). The `:f defn myFindExample` command will return only the second result (on line 38).

Further more, `:f myFindDataExample` returns 3 occurrences:

```
UserGuideExamples: (line 43, column 13) -
UserGuideExamples.MyFindDataExample - data public MyFindDataExample =

UserGuideExamples: (line 45, column 14) -
UserGuideExamples.MyFindDataExample - deriving Eq, Ord;

UserGuideExamples: (line 45, column 18) -
UserGuideExamples.MyFindDataExample - deriving Eq, Ord;
```

`:f instByType MyFindDataExample` returns the latter two occurrences for the classes it derives. The`:f instByClass Ord` command returns the following amongst results from other modules:

```
UserGuideExamples: (line 45, column 18) - Cal.Core.Prelude.Ord - deriving
Eq, Ord;
```

Similarly, `:f constructions Boolean` returns the following amongst results from other modules:

```
     UserGuideExamples: (line 41, column 34) - Cal.Core.Prelude.True -
     myFindRefExample = myFindExample True;
```

3.2 Single-threaded Debugging

Here is an example of a single threaded debugging session. Make sure the property `org.openquark.cal.machine.debug_capable` is defined.

In section 2.3, `UserGuideExample.cal` with a function `myFactorial` was created. Switch to the module by using the `:sm UserGuideExample` command.

First, set a break point on the method:

```
UserGuideExamples>:bp UserGuideExamples.myFactorial
      Breakpoint enabled for UserGuideExamples.myFactorial
```

`:bp clear` will remove all the breakpoints. To view the list of breakpoints:

```
UserGuideExamples>:bp show
      Breakpoints:
          UserGuideExamples.myFactorial
```

Then evaluate the expression:

```
UserGuideExamples>myFactorial 5
      running: UserGuideExamples.myFactorial 5
      Execution suspended in: UserGuideExamples.myFactorial
      Argument values:
       myFactorial$n$1 - 5
```

Now the program suspends and you can view some properties of the state, for instance the function that is being suspended and the argument names

```
UserGuideExamples.myFactorial#>:show function
      UserGuideExamples.myFactorial

UserGuideExamples.myFactorial#>:show argNames
      myFactorial$n$1 - Cal.Core.Prelude.Integer

UserGuideExamples.myFactorial#>:show stack
      org.openquark.cal_UserGuideExamples.My_Factorial.f1S(My_Factorial.java)
      org.openquark.cal_UserGuideExamples.Iceruntarget.f(Iceruntarget.java)
      org.openquark.cal_Cal_Core_Prelude.Id.f(Id.java)
```

Now continue the execution:

```
UserGuideExamples.myFactorial#>:step
      Resuming execution of: UserGuideExamples.myFactorial 5 on thread:
      MainIceRunThread
      MainIceRunThread> Cal.Collections.List.product
      Cal.Core.Prelude.$dictCal.Core.Prelude.Num#Cal.Core.Prelude.Integer
      Execution suspended in: Cal.Collections.List.product
      Argument values:
          $dictvarCal.Core.Prelude.Num#16 -
      Cal.Core.Prelude.$dictCal.Core.Prelude.Num#Cal.Core.Prelude.Integer
```

Turn on the trace function, and then set some filters to remove excess information

```
Cal.Collections.List.product#>:trace on
      General function tracing enabled.

Cal.Collections.List.product#>:step
      Resuming execution of: UserGuideExamples.myFactorial 5 on thread:
      MainIceRunThread
      MainIceRunThread>
      Cal.Core.Prelude.$dictCal.Core.Prelude.Num#Cal.Core.Prelude.Integer 12
      Execution suspended in:
      Cal.Core.Prelude.$dictCal.Core.Prelude.Num#Cal.Core.Prelude.Integer

Cal.Core.Prelude.$dictCal.Core.Prelude.Num#Cal.Core.Prelude.Integer#>:fa
Boolean
```

Restart the program for the new filter to be effective

```
Cal.Core.Prelude.intToInteger#>:re
UserGuideExamples>myFactorial 5
```

Now you will see the tracing details have been filtered out unless they contain Boolean in it. Use `:fs` to view the list of filters and `:fr <index>` to remove them.

3.3 Multi-threaded Debugging

To support multithreaded debugging, system properties `org.openquark.cal.machine.lecc.concurrent_runtime` and `org.openquark.cal.machine.debug_capable` should be defined. The example is based on a module named `ParallelList_Tests.cal` within the `cal.libraries.test.cws` workspace.

First, we need to add `CAL_Libraries` project to the class path for ICE. On the Eclipse menubar, select the Run menu → Open Run Dialog… → Select the ICE session → open the Classpath tab → Select User Entries → Click on the "Add Projects…" button → Select `CAL_Libraries` project and select "Add required projects of selected projects".

Under the User Entries item above, ensure that log4j.jar appears in the tree. If it is missing, click on the "Add JARs…" button. Go through the directories to select "`\import\Titan_Research\win32_x86\release\bin\External\java\log4j.jar`".

First, switch to the workspace and load the module

```
Cal.Test.Core.Prelude_Tests>:ldw cal.libraries.test.cws
      CAL: Compilation successful
      CAL: Finished compiling in 2908ms
      CAL: 156 modules loaded (156 as dynamic, 0 as static).
```

```
Cal.Test.Core.Prelude_Tests>:sm ParallelList_Tests
```

We need to run the function then suspend it by immediately pressing Enter a second time (right after the Enter for running the function)

```
Cal.Test.Experimental.Concurrent.ParallelList_Tests>processSlowItems
Parallel.threadPerTaskExecutor 20
```

Now you should see some output similar to this:

```
      Running:
      Cal.Test.Experimental.Concurrent.ParallelList_Tests.processSlowItems
      Cal.Experimental.Concurrent.Parallel.threadPerTaskExecutor 20

      Execution suspended in: Cal.Core.Prelude.outputCalValueStrict
      Argument values:
          outputCalValueStrict$calValue$1 - 3
```

To view the threads being suspended:

```
(Thread-22) Cal.Core.Prelude.outputCalValueStrict#>:threads
      Currently suspended threads:
      42: MainIceRunThread
      43: Thread-21
      44: Thread-22 <- current thread
      46: Thread-24
      48: Thread-26
      50: Thread-28
```

We will resume the current thread only, and a subsequent `:threads` command shows that Thread-22 was no longer in the list.

```
(Thread-22) Cal.Core.Prelude.calValueToObject#>:ret
      Resuming execution of:
      Cal.Test.Experimental.Concurrent.ParallelList_Tests.processSlowItems
      Cal.Experimental.Concurrent.Parallel.threadPerTaskExecutor 20 on
      thread: Thread-22
      Execution suspended in: Cal.Core.Prelude.outputCalValueStrict
      Argument values:
          outputCalValueStrict$calValue$1 - 3

 (Thread-28) Cal.Core.Prelude.outputCalValueStrict#>:threads
      Currently suspended threads:
      42: MainIceRunThread
      43: Thread-21
      46: Thread-24
      48: Thread-26
      50: Thread-28 <- current thread
```

Let’s switch to the main thread which has index 42.

```
(Thread-28) Cal.Core.Prelude.outputCalValueStrict#>:thread 42
(MainIceRunThread) Cal.Core.Prelude.outputInt#>
```

Stepping only affects the current thread.

```
(MainIceRunThread) Cal.Core.Prelude.outputInt#>:step
```

```
      Resuming execution of:
      Cal.Test.Experimental.Concurrent.ParallelList_Tests.processSlowItems
      Cal.Experimental.Concurrent.Parallel.threadPerTaskExecutor 20 on
      thread: MainIceRunThread
      Execution suspended in: Cal.Core.Prelude.jList_add
      Argument values:
          $x0 - []
          $x1 - 5
```

Resume execution on all threads. Voila, we get a result.

```
(MainIceRunThread) Cal.Core.Prelude.jList_add#>:re
      Resuming execution of:
      Cal.Test.Experimental.Concurrent.ParallelList_Tests.processSlowItems
      Cal.Experimental.Concurrent.Parallel.threadPerTaskExecutor 20
      Run Time:        1,138,420 ms

      Output:
[5, 3, 0, 8, 0, 6, 2, 3, 9, 0, 3, 6, 1, 1, 9, 4, 8, 1, 8, 0]
```

4 Command Reference

Commands start with colon (:) and can be entered directly onto the command line. In the following description, angle brackets (< >) represent tokens that should be replaced with the appropriate variable. Square brackets ([ ]) signify optional arguments or alternative command names. Commands are case insensitive and grouped according to their functionality.

4.1 General

`:q[uit]`

End the current ICE session.

`:v[ersion]`

Display the version of the CAL platform in use.

`:rs  [module name]`

Reset any cached CAFs in the named module and its dependant modules. If none specified, memorized results in all modules will be discarded. Partially qualified names are accepted.

`:rsm [module name]`

Reset machine state, including CAFs, associated with the named module and any dependant modules or all modules if none specified. Partially qualified names are accepted.

`:rc`

Recompile. This will only compile modified modules and will only regenerate code for modified CAL entities. (See Table 1)

`:rca`

Recompile all. This will compile all modules, but only regenerate code for modified CAL entities. (See Table 1)

`:rl`

Reload the program. This will recompile and regenerate all CAL entities. (See Table 1)

Table 1. Difference in compilation and code regeneration between `:rc, :rca` and `:rl`.

Command	Action	Modified CAL entities	All CAL entities
`:rc`	Compile Regenerate code	Yes Yes	No No
`:rca`	Compile Regenerate code	Yes Yes	Yes No
`:rl`	Compile Regenerate code	Yes Yes	Yes Yes

:sm <target_module_name>

Set the module to operate in. The target name can be either fully or partially qualified since partially qualified module names will be resolved. For example, "`:sm M2`" can be used to change to the Cal.Test.General.M2 module.

:rr

Re-evaluate the last entered CAL expression. If there is no previous expression evaluated, an error message will be displayed.

:script <script file name>

Run the script in the file specified by its full path name.

:of <output file name>

Set a file location where subsequent output will be mirrored. In other words, change the output file so any output generated after this command will be directed to the specified file.

:h[elp] [<topic>]

Show the ICE help with valid command names and corresponding descriptions. If the topic is specified, only commands within the group will be displayed. Otherwise, general and history commands will be displayed, as well as a list of the command topics. The help function is case insensitive and it recognizes unambiguous partial topic names or it displays the set of possible topics. For example, typing `:h oth` or `:h OTH` will have the same effect as `:h other`, but `:h o` will cause "`Possible help topics: options, other`" to be displayed. Use `:h all` to display all the commands.

Table 2. Help topics and descriptions.

Topics	Description
`:h all`	show all commands
`:h general`	general commands
`:h info`	commands to describe the current environment
`:h find`	find entities in the workspace
`:h workspace`	program and workspace manipulation
`:h debug`	debugging commands
`:h benchmark`	benchmarking commands
`:h CALDoc`	generate and show CALDoc
`:h refactor`	commands to refactor the source code
`:h utilities`	utilities to generate code, cal archives, etc.
`:h history`	commands to show or execute previous commands
`:h options`	command line options and environment setting help
`:h other`	other commands

4.2 Info

`:smd <dependee_module_name>`

Show all modules which depend on the named module either directly or indirectly.

`:si  <dependent_module_names>`

Show all modules that the named modules import either directly or indirectly. Multiple modules can be specified with a space in-between each name. Partially qualified module names are accepted. Statistics are displayed including the number of lines of the dependent modules, number of lines for each dependee module, as well as the ratio of dependee lines to dependent lines.

Note that if both module A and B are specified and A depends on B, B will not be listed as a dependee module. Thus it is possible that the number of dependee modules for A alone is greater than that of A and B combined.

`:t <expression>`

Display the CAL type of the given expression in fully qualified form. For example, `:t 5.0` will return `Cal.Core.Prelude.Double`, whereas `:t 5` will return `Cal.Core.Prelude.Num a => a` since the CAL compiler cannot inferred the type for the given expression.

`:sps [module names]`

Show summarized program statistics for the named module(s), or all modules if none specified. Statistics are in areas of functions, constructors, type classes, instances and instance methods. If multiple modules are named, the combined statistics are shown.

`:ss`

Display the current ICE and environment settings, similar to the displayed message shown during ICE start up.

`:sts`

Show the current settings for function tracing. The build needs to be trace capable by setting the system property `org.openquark.cal.machine.debug_capable`.

`:swi`

Show debug information about the workspace, including workspace declaration and its location, workspace location and workspace contents. Workspace contents will include all the modules in the workspace and their file locations.

`:lcfs`

Loaded Class File Size. This shows the number and size of classes dynamically loaded by the CAL class loader. This only supported by the LECC machine.

`:cfs`

Class File Size. Show the total size and count of all class files constituting the compiled Java form of the CAL entities in the current workspace. This only supported by the LECC machine in source or static bytecode mode.

`:lcafs [<module_name>] [<includeDependees>]`

Lists all the constant applicative forms (CAFs) in the given module and any dependee modules as indicated with true or false. The target module and false are the default values if no arguments were specified.

4.3 Find

Find commands allow you to locate CAL entities within the workspace.

`:f[ind] [all] <entity_name>`

Display the locations of each occurrence of the specified function, method, or data constructor

`:f[ind] ref <gem_name>`

Display the locations of each reference to the specified function, method, or data constructor

`:f[ind] defn <gem_or_type_class_name>`

Display the location where the specified function, method, data constructor, type, or type class is defined

`:f[ind] instByType <type_name>`

Display the locations of each instance definition for every class that the specified type is an instance of

`:f[ind] instByClass <type_class_name>`

Display the locations of each instance of the specified type class

`:f[ind] constructions <type_or_constructor_name>`

Display the locations where values are constructed. If a type constructor is specified then any construction using a data constructor of that type is found.

Table 3. Summary of various `:find` commands and their arguments.

First argument	Look for	Second argument
`[all]`	Occurrences	Function, method, data constructor
`Ref`	References	Function, method, data constructor
`defn`	Definition	Function, method, data constructor, type, type class
`instByType`	Instance definition	Type class
`instByClass`	Instances of type class	Type class
`constructions`	Construction of values	Type constructor, type

4.4 Workspace

`:lmw <dependent_module_names>`

Load the minimal workspace that contains all of the named modules and their dependee modules. Partially qualified module names are accepted. The named modules have to be available within the current workspace. A minimal workspace is created, compiled and loaded. If the current working module is not in the new minimal workspace, the first named module will be set as the new working module. To save a minimal workspace to file, refer to `:wsi` command in Utilities.

`:adm <module_name>`

Add Discoverable Module. Load a module from the current environment where it is unambiguous to the current workspace. The workspace will be recompiled if the addition was successful. Partially qualified names are accepted.

`:ldw <workspace_file_name>`

Load Discoverable Workspace. Load a workspace file from the current environment. ICE will switch to another working module if the current module is not in the new workspace.

`:lw  <workspace_file_name>`

Load workspace from the specified file. The full path to the file should be given. The new workspace is initialized and the working module is set to a default module.

`:lm`

List the available modules in the current workspace.

`:sdw`

Show the names of the workspace files found in the current environment in sorted order.

`:sdm`

Show the names of the modules found in the current environment, but not in the current program.

`:rm  <target_module_name>`

Remove the named module and any dependent modules from the current workspace. If the current module is removed then a default module will be set as the working module. Partially qualified names are accepted.

4.5 Debug

To use the debug tool, the property `org.openquark.cal.machine.debug_capable` must be defined. Multiple threads in the same CAL execution are handled if `org.openquark.cal.machine.lecc.concurrent_runtime` is defined. The debug tool allows the user to set break points on CAL functions. When a break point is reached, the program is suspended for the user to inspect the state of the machine with various `:show` commands on the current thread. At this time the user is also able to change function tracing settings if enabled and execute a subset of the regular ICE commands. The user can either resume execution on a single thread or the entire program or step to the next function on the current thread.

Basic Operation

`:step`
Step through a suspended CAL program to the next function on the current thread.

`:step <CAL expression>`
Step through the provided CAL expression.

`:re`
Resume execution of suspended CAL program (all threads).

`:ret`
Resume execution of the current suspended thread.

`:sus[pend]`
Suspend all threads in the executing CAL program.

`:threads`
Show the current suspended threads.

`:thread <thread id>`
Set the current suspended thread.

`:te`
Terminate execution of suspended CAL program (all threads).

Show State of Machine

`:show`
Show the state of the current CAL suspension associated with the current thread. This displays the function name and the names and values of any arguments.

`:show function`
Show the name of the suspended function.

`:show argnames`
Show the names of the arguments to the suspended function.

`:show argTypes`
Show the types of the arguments to the suspended function.

`:show <argument name>`
Show the value for the named argument.

`:show stack [expanded]`
Show the call stack of CAL functions. If expanded is specified the call stack will include non-CAL functions.

Breakpoint

`:bp <qualified name>`
Toggle a breakpoint for the named CAL function. Partially qualified names are accepted.

`:bp show`
Show the current set of breakpoints.

`:bp clear`
Clear all breakpoints.

Trace

`:trace on`
Enable tracing for all CAL functions.

`:trace off`
Disable tracing for all CAL functions.

`:trace <qualified name>`
Toggle tracing for the named CAL function. This overrides the general trace setting. Partially qualified names are accepted.

`:trace clear`
Clear all CAL functions for which tracing had been specifically enabled.

`:trace arguments on`
Enable tracing of function arguments.

`:trace arguments off`
Disable tracing of function arguments.

`:trace threadname on`

Include the thread name in trace messages.

`:trace threadname off`

Exclude the thread name in trace messages.

`:fa <regExpr>`

Add the given filter to the list of filters in effect for the function trace output. The new filters are effective upon rerun of the function.

`:fr <index>`

Remove the indicated filter from the filter list for function tracing.

`:fs`

List the filters for the function trace output currently in effect.

Others

`:stt`

Display the stack trace for an error result.

`:d <function_name>`

Display the disassembled form of the named function. Disassembly is only supported by G-machine.

`:da`

Display the disassembled form of all functions. Disassembly is only supported by G-machine

`:ded`

Toggle execution diagnostics.

4.6 Benchmark

`:pt <cal code>`

Run a performance test on the specified CAL code. The performance test involves evaluating the expression a number of times, where the number can be set via the `:sptr` command. The individual time for each run, the average run time and the standard deviation are displayed. The average is calculated after discarding the initial run due to potential overhead from class loading and code generation.

:ptm <cal code>

Run a performance test on the specified CAL code similar to `:pt`. However, the machine state is reset prior to each run in the test. In addition to individual time for each run, the average run time and the standard deviation, machine statistics for each run are displayed.

:brf [benchmark file name]

Set a file where benchmark results from `:pt` and `:ptm` commands will be written to. If no name is provided then the current file is simply closed. The file content can be manipulated by a set of CAL functions in the Benchmarks module. Also, when the file is set successfully, the benchmark results label will be changed to the file name.

:brl <benchmark label>

Set a label for runtimes recorded in the benchmark results file. Resetting the results file will overwrite the existing benchmark label to the file name.

:sptr <number of runs>

Set the number of runs used in performance testing. Note that the initial run is always discarded from the average due to potential overhead from class loading and code generation.

4.7 CALDoc

:docgen <options>

Run the CALDoc HTML documentation generator as specified by command line arguments. Documentations are generated to a default location unless specified. `:docgen -help` will display all the options. If parsing of the command line arguments fails, then the command line usage info is logged to the logger.

Table 4. Options for the documentation generator.

Options	Description
`-d <directory>`	Destination directory for output files
`-public`	Show only public entities (default)
`-private`	Show all entities - public, protected and private
`-modules <regexp>`	Include only the modules matched by the regular expression
`-excludeModules <regexp>`	Exclude the modules matched by the regular expression
`-entities <regexp>`	Include only the entities (functions, types and data constructors, classes and class methods) matched by the regular expression
`-excludeEntities <regexp>`	Exclude the entities (functions, types and data constructors, classes and class methods) matched by the regular expression
`-metadata [override]`	Include metadata in documentation (use the 'override' option to hide CALDoc that is superceded by metadata)

`-author`	Include author information
`-version`	Include version information
`-qualifyPreludeNames`	Qualify the names of Prelude entities outside the Prelude module
`-use`	Create usage indices
`-doNotSeparateInstanceDoc`	Do not separate instance documentation from the main documentation pages
`-locale <locale>`	Locale to be used, e.g. en_US or fr
`-windowTitle <text>`	Browser window title for documentation
`-docTitle <html-code>`	Include title for overview page
`-header <html-code>`	Include header text for each page
`-footer <html-code>`	Include footer text for each page
`-bottom <html-code>`	Include bottom text for each page
`-verbose`	Output detailed messages about what the tool is doing
`-quiet`	Do not display status messages to screen
`-help`	Display command line options and exit
Non-standard -X options: `-XexcludeTestModules`	Exclude 'test' modules

`:docm <module_name>`

Show the CALDoc comment associated with the named module. Partially qualified module names are accepted.

`:docf <function_or_class_method_name>`

Show the CALDoc comment associated with the named function or class method.

`:doct <type_name>`

Show the CALDoc comment associated with the named type constructor.

`:docd <data_constructor_name>`

Show the CALDoc comment associated with the named data constructor.

`:docc <type_class_name>`

Show the CALDoc comment associated with the named type class.

`:doci <type_class_name> <instance_type_constructor>`

Show the CALDoc comment associated with the named instance.

`:docim <method_name> <type_class_name> <instance_type_constructor>`

Show the CALDoc comment associated with the named instance method.

4.8 Refactor

`:rf imports [<module_name>] [nogroup]`

Clean the import declarations in the specified module or in the current module if none specified. If the `nogroup` argument is specified, `using` items will not be grouped together or reordered (although the names within the items are still reordered).

`:rf typeDecls [<module_name>]`

Add type declarations to functions and local definitions that currently lack them in the specified module or in the current module if none specified. A summary of the refactoring will be displayed.

`:pp [<module_name>] [<function_name> [<function_name> ...]] [startLine endLine]`

Pretty Printer formats the module specified or the current module if none specified. Optional arguments can be used to specify the function(s) to format or the line range to format.

`:ren[ame] [<category>] <old_identifier> <new_identifier>`

Rename `old_identifier` to `new_identifier`. `Category` is one of function, dataConstructor, typeConstructor, typeClass or module. The category should be specified if the old identifier is ambiguous. Furthermore, the new identifier must be unique and of the same type as the old identifier (ie, same case for the first letter). Identifiers other than module names should be fully qualified.

4.9 Utilities

`:jar [-verbose] (-main <functionName> <mainClassName> | -lib <moduleName> <libClassScope> <libClassName>)+ <outputJarName> [-src <outputSrcZipName>]`

Create a standalone JAR from a given entry point.
`-verbose` is an option flag for displaying more detailed status information
`-main` specifies a CAL application to be included with the standalone JAR
- `functionName`: should be fully qualified CAL function name which is the main entry point for the application
- `mainClassName`: is the fully qualified name of the Java main class to be generated

`-lib` specifies a CAL library to be included with the standalone JAR
- `moduleName`: fully-qualified name of the CAL library module
- `libClassScope`: scope of the generated library class. Can be one of public, protected, package or private.
- `libClassName`: fully-qualified name of the Java main class to be generated

`outputJarName` is the name of the output JAR file or a path.
`-src <outputSrcZipName>` is optional for specifying the name of the output zip file containing source files for the generated main classes and library classes.

`:car [-keepsource] [-nocws | -nosuffix] [-s] [-jar] [-d] <output directory>`

Create a CAL Archive (Car) file from the current workspace in the given directory.
`-keepsource` specifies the generated Cars will contain the full source of the modules, rather than the default sourceless modules with empty CAL file stubs.
`-nocws` specifies a new workspace declaration file will not be generated for each output Car.
`-nosuffix` specifies the workspace declaration files generated will end simply with .cws rather than .car.cws.
`-s` specifies the modules that come from Cars will be skipped over in the output Cars.
`-jar` specifies Car-jars will be generated instead of regular Car files. In this case, the output files will be placed directly under the specified output directory, rather than in a Car subfolder. If `-nocws` is not specified, then the new workspace declaration file will be packaged *inside* the Car-jar
`-d` specifies a Car file is created in the given directory for every workspace declaration file imported by the current workspace.

`:wsi [-l] <dependent_module_names> "<workspace_file_name>"`

Create a minimal workspace that contains all of the named modules and save it to workspace_file_name. If -l is specified, then the newly created minimal workspace will be loaded. Fully-qualified module names should be used.

`:gen[erate] deriving [compact] <type_name>`

Generate and display instance definitions for the given type for the classes Eq, Ord and Show, such as these definitions can be copy-pasted into a module and then compiled. The default implementation is compact, in that notEquals is defined in terms of equals, and the Ord methods are defined in terms of compare.

The command ignores any existing deriving clauses on the type and generates the class instances. The generated function names are not guaranteed to not conflict with existing functions in the module. Also, it does not check whether the type is valid for deriving

`:gen[erate] deriving efficient <type_name>`

Generate and display instance definitions for the given type for the classes Eq, Ord and Show, such as these definitions can be copy-pasted into a module and then compiled. The implementation is efficient, in that the Eq and Ord methods are all implemented directly using case expressions.

`:javaBinding <module name pattern> <target package> [internal] [test]`

Generate the Java binding class for the modules which match the given regular expression. The target package specifies the package which will contain the generated binding class. By default bindings are generated only for public members of the module. Specifying the `internal` parameter will generate bindings for only the non-public members of the module. Specifying the `test` parameter will check that a valid and up-to-date binding class exists, without actually generating the class.

4.10 History

`:spc`

Show previously entered commands and expression evaluation in a numbered list in order of execution., `:pc`, `:spc` and unrecognized commands will not be added to the list. Also, a command is only added if it is not already in the list. Recognized commands with invalid arguments will be added to the list. The list of commands is retained for the entire ICE session regardless of the current workspace and working module.

`:pc <command number>`

Execute a previous command indicated by the command number. If the selected expression evaluation is undefined in the current module, a compilation error occurs. If there are duplicate function names within the 2 modules, the one for the current module will take precedence.

`:npc <number of commands>`

Set the size of the command history list specific to this ICE session. The default of any ICE session is 10. If the current command list is greater than the new size specified, the most recent commands are kept in the list.

4.11 Other

`:caf <CAF name>`

Directly execute the named constant applicative form (CAF).

`:mt <machine_type>`

Set the runtime machine type. If an invalid name is provided, a list of valid machine types will be displayed. Note that the argument is case sensitive.

`:sms`

Toggle displaying incremental machine stats from evaluating CAL expressions.

`:so`

Toggle displaying output from CAL expressions.