CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
LABORATOIRE d'ANALYSE et d'ARCHITECTURE des SYSTÈMES

7, Avenue du Colonel Roche, 31077 Toulouse Cedex, France
Tel: (+33) 61 33 62 00. Telex: LAASTSE.520930F. Telefax: (+33) 61 55 35 77

LCS USERS MANUAL

Version 5.0 of June1998

Release Notes
Language Reference
User Interfaces
Libraries

Bernard Berthomieu

Bernard.Berthomieu@laas.fr
tel: +33/0 5 61 33 63 63
fax: +33/0 5 61 33 64 11

LAAS/CNRS Report 98269, June 1998

Libraries

Contents

1 Organization of libraries

Organization

The LCS libraries provide:

• A number of *structures* physically found in files in directory `lib/structures` of the distribution. These structures are logically organized into four (super-)structures, referred to as *libraries*: `Core`, `Prelude`, `Extras`, and `Network`, all to be described. Though there is no technical necessity for this, each file in `lib/structures` defines a single structure (possibly containing itself some sub-structures) with the same name than the file prefix (`.sml` or `.lcs` omitted) but with the first letter capitalized (e.g. file `lib/structures/array.sml` defines structure `Array`). Reading such a file reads the structure it contains at lcs toplevel, but it does NOT open the structure. Further, structure files do not modify fixity or overloading status of identifiers.

• A number of installation *scripts*, found in files in directory `lib/scripts` of the distribution. All scripts begin with letter "`l`" (for "load"). Scripts are sequences of toplevel declarations reading and installing a number of structures; there are currently scripts for loading and installing the `Prelude`, `Extras`, and `Network` libraries. Of course, one may still load a particular structure from `lib/structures` without having to load and install a full library, but in that case, installation of the structure is under the user's responsibility: Structures like `Int`, `Mp`, `Real`, `Rat`, and possibly others, are designed with overloading or infixing of some operators in mind, just opening these structures would not overload or infix these operators (structures do not capture such information). In addition to opening a number of structures, installation scripts set infix status for a number of operators and take care of overloading of those operators intended to be overloaded (typically `print`, `makestring`, and arithmetic and comparison operators on the various constants).

• A `lib/local` directory. This directory is empty in the distribution, it may be used to store site-specific structure and/or script files, or compatibility files with other ML implementations.

• Two more directories, named `lib/demos`, and `lib/examples`, hold what their names suggest. Their contents may vary with the distribution.

Library files, whether containing structures or scripts, have either extension `sml` or `lcs`. Files with extension `sml` contains code making use of SML only features (behaviors excluded, thus), those with extension `lcs` rely on lcs behaviors. The former may be loaded in lcs with any process option, the latter should not be loaded when process option "-n" is used since that option precisely disables behavior features. Library files are read in by one of the commands `read_file`, `use`, `needs`, or `incl` (see structure `System` in Section 2.2 for their differences), these functions make use of a search path to locate files (see `System.search_path`), this search path initially include directories `<lib>/structures`, `<lib>/script`, and `<lib>/local`, where `<lib>` is the installation path of the lcs libraries (passed by the lcs command, see the *Release Notes* for the installation procedure).

Libraries

The four packaged libraries organize structures from least specialized to most specialized:

• Library `Core` contains all lcs primitives, it is required to build the other libraries. It is mentioned here for completeness but it is not documented. It is built-in.

• Library `Prelude` is initially loaded and installed. Its description is provided in Section 2.2 for documentation purposes. It relies on the `Core` structure, which it strictly extends. It includes all Core SML bindings. Assuming structure `Core` has not been rebound, the `Prelude` library can be rebuilt and reinstalled by reading script `lprelude`, described in Section 3.

• Library `Extras` provides additional general purpose types, functions, and behaviors. It is not initially loaded, it is loaded and installed by reading script `lextras`.

• Library `Network` provide structures specific for interconnection of lcs applications and distributed programming. It is not initially loaded either, it is loaded and installed by reading script `lnetwork`.

No symbol initially has overloading status, identifiers `~ + - * div mod divide < > <= >= abs min max pow print makestring` are initially multi-valued. Symbols which are given infix status in script `lprelude` (in addition to “=”) initially have infix status, these are `* div / pow + - ^ :: @ <> < <= > >= = o := ^^`.

2 structures

2.1 Core: lcs primitives

Structure `Core` holds all primitives of lcs. It is mentioned for completeness but is not documented. It is required to define the `Prelude`, `Extras`, and `Network` structures described in the following sections.

Core

CONTENTS

All primitives of lcs.

SIGNATURE

```
structure Core = ...
```

DESCRIPTION

(omitted)

EXAMPLES

SEE ALSO

2.2 Prelude: initial environment

This section describes structure `Prelude` and all of its sub-structures. Structure `Prelude` strictly extends the `Core` structure; it is initially loaded and installed by script `lprelude` (see Section 3). It can be reloaded and reinstalled by reading in that script.

Array

CONTENTS

Type array and array primitives.

SIGNATURE

```
structure Array =
   datatype 'a array   ['a array]
   exception Size : exn
   val array : int * '1a -> '1a array
   val length : 'a array -> int
   exception Subscript : exn
   val sub : 'a array * int -> 'a
   val update : 'a array * int * 'a -> unit
   val tabulate : int * (int -> '1a) -> '1a array
   val arrayoflist : '1a list -> '1a array
   val listofarray : 'a array -> 'a list
```

DESCRIPTION

`'a array`	Type of arrays, or homogeneous mutable products. Their component can be accessed by indexing. Arrays admit equality: two arrays are equal if both are empty or they both result of the same call of primitive `array`.
`[||], [|v1,...|]`	Arrays are represented like lists, but using brackets "`[|`" and "`|]`". Arrays are not allowed in patterns.
`array (n,v)`	Produces an array with `n` fields, each with content `v`. Raises exception `Size` if n<0.
`length a`	Returns number of components of array `a`.
`sub (a,i)`	Returns component at index `i` in `a`; indices range from `0` to (`length a - 1`). Raises exception `Subscript` if index out of range.
`update (a,i,v)`	Replaces component `i` of array `a` by `v`. Raises exception `Subscript` if index `i` is out of range.
`tabulate (n,f)`	Produces an array of length `n` with (`f i`) as i^{th} component, for i in `0..n-1`. Raises exception `Size` if `n<0`.
`arrayoflist l`	Builds an array with its components taken in list `l`.
`listofarray a`	Reverse of above; builds a list from the components of array `a`.

EXAMPLES

SEE ALSO

Structures `Prelude.Bytearray`, `Prelude.Vector`, `Prelude.String`.

Beh

CONTENTS

Behavior types and some frequently needed agent combinators.

SIGNATURE

```
structure Beh =
   val ^^ : {out#’’a:’b}_c * {inp#’’a:’b, tmp#’’d:’e}_c -> {tmp#’’f:’g}_c
   val NET : (_a * _a -> _a) -> ('b -> _a) -> 'b list -> _a
   val DUP : (_a * _a -> _a) -> _a -> int -> _a
   val PAR : ('a -> _b) -> 'a list -> _b
   val SUM : ('a -> _b) -> 'a list -> _b
   val PIPE : ('a -> {inp#''b: 'c, out#''b: 'c}_d) -> 'a list ->
              {inp#''b: 'c, out#''b: 'c}_d
   val INP : ('a list -> {inp: 'a}_b) -> int -> {inp: 'a}_b
   val OUT : {out: 'a}_b -> 'a list -> {out: 'a}_b
   val attach : window -> _a -> _a
   val control : _a -> {dis}_a
   val lcs : _a
   val wlcs : window -> _a
```

DESCRIPTION

<behavior types>
: Types of behaviors. Neither behavior types nor behavior constructions are expressible in terms of other types. Behaviors admit recursion and do not admit equality.

/\, \/, etc
: Concurrency and ambiguity behavior combinators can be used as functions as well, when prefixed by keyword **op**. They become strict functions then. Syntactically, application of such combinators in functional form are no more considered behavior expressions. So e.g. the following declaration is illegal (illegal recursion) though the same with /\ in infix form is legal:

```
agent a = (op /\) (stop,stop);
```

DUP f a n
: Iterates n times behavior a by combinator f: i.e.

```
DUP f a n = f a (f a (... a))
```

NET f a l
: Like DUP except that a is a parameterized behavior; the behaviors composed take their parameters in list l:

```
NET f a [l1,...,ln] = f (a l1) (f (a l2) (...(a ln)))
```

PAR, SUM, PIPE
: Specializations of NET:

```
PAR a [l1, ..., ln] = a l1 /\ a l2 /\ .. /\ a ln
SUM a [l1, ..., ln] = a l1 \/ a l2 \/ .. \/ a ln
PIPE a [l1, .., ln] = a l1 ^^ a l2 ^^ .. ^^ a ln
```

INP, OUT
: Allow iterated input and output, they obey:

```
OUT a [l1, ..., ln] = out!l1=>out!l2=>.. =>out!ln=> a
INP a n = inp?x1=>inp?x2=>.. =>inp?xn=> a [x1,x2,.. ,xn]
```

control a
: Behaves as a except that terminates if it accepts communication on term

`attach w a`	Behaves as a except that terminates if ^C is typed in window w.
`lcs ()`	Opens a window with `Io.open_win` and starts a second lcs toplevel in it. Proper operation of the new toplevel requires process option -s (otherwise the two toplevels can't be distinguished).
`wlcs w`	As `lcs`, except the second toplevel operates on the window `w` passed as argument.

Note: Combinators are intended to help development of complex programs. A simple naming discipline for ports helps remembering their interfaces: all of their input ports are prefixed by "`inp`", and all of their output ports are prefixed by "`out`". One may use relabelling to change these names to make them match those expected in a particular context.

EXAMPLES

```
- fun from (n:int) = out!n => from (n+1);
val from = - : int -> {out: int}_a

- agent sq = inp? (x:int) => out! (x*x) => sq;
val sq = - : {inp: int, out: int}_a

- fun sqfrom n = from n ^^ sq;
val sqfrom = - : int -> {out: int}_a

- agent cmp = inp1? x => inp2? y => out! (x=y) => stop;
val cmp = - : {inp1: ''a, inp2: ''a, out: bool}_b
```

(* Agents `lcs` and `wlcs` are bound to the lcs top-level behavior. The strong process option ("-s") allows one to open several independent toplevels working on the same lcs machine. Call "lcs -s" and try for instance: *)

```
- sstart lcs;
val it = - : _a
```

SEE ALSO

`Extras.Timing`: timing facilities for behaviors.

`Extras.Dataflow`, `Extras.Broadcast`, `Extras.Parrays`, `Extras.Pio`: various other process combinators.

Bool

CONTENTS

Type bool, boolean constants and connectives, equality predicate.

SIGNATURE

```
structure Bool =
   datatype bool   [bool]
   con true : bool
   con false : bool
   val = : ''a * ''a -> bool
   val <> : ''a * ''a -> bool
   val not : bool -> bool
```

```
    val andalso : bool * bool -> bool
    val orelse : bool * bool -> bool
    val exists : ('a -> bool) -> 'a list -> bool
    val for_all : ('a -> bool) -> 'a list -> bool
    exception Bool_of_string : exn
    val bool_of_string : string -> bool
    val string_of_bool : bool -> string
    val makestring : bool -> string
    val print : bool -> bool
```

DESCRIPTION

`bool`	Boolean type. Admits equality. Booleans constants and not could be defined by: **`datatype`** bool = true | false
`true, false`	Boolean constructors. Admitted in patterns.
`=, <>`	Equality (=), and inequality (<>). The equality predicate cannot be expressed in terms of existing functions.
`not`	Negation function.

`andalso, orelse`	Conjunction (**`andalso`**) and disjunction (**`orelse`**) are non strict when used in infix form, but strict when used in functional form (preceded by **`op`** then). When in infix form, **`andalso`** and **`orelse`** are expanded as follow at parse-time:
	`b1` **`andalso`** `b2 ==` **`if`** `b1` **`then`** `b2` **`else`** `false`
	`b1` **`orelse`** `b2 ==` **`if`** `b1` **`then`** `true` **`else`** `b2`

`exists, for_all`	List iterated boolean connectives.
`string_of_bool b`	String representation of a value v of type unit.
`bool_of_string`	`Returns` the value of type bool represented by string s. Raises exception `Bool_of_string` if s does not represent a value of type `bool`.
`makestring`	Makes a string from a value of type unit; `makestring` is overloaded at all constant types at top-level.
`print`	Identity function; prints string representation of b on the standard terminal output.print is overloaded at all constant types at top-level.

EXAMPLES

SEE ALSO

Bytearray

CONTENTS

Arrays of bytes.

SIGNATURE

```
structure Bytearray =
    datatype bytearray   [bytearray]
    exception Size : exn
    val bytearray : int * int -> bytearray
    val bytearray2 : int -> bytearray
```

```
   val length : bytearray -> int
   exception Subscript : exn
   val sub : bytearray * int -> int
   val update : bytearray * int * int -> unit
   val bytearrayoflist : int list -> bytearray
   val listofbytearray : bytearray -> int list
   val bytearrayofstring : string -> bytearray
   val stringofbytearray : bytearray -> string
```

DESCRIPTION

`bytearray`	Arrays of bytes. Bytes are represented by integers in the range 0..255. Bytes are represented by integers in the range 0..255. Bytearrays are created by one of bytearray or bytearray2:
`bytearray (n,v)`	Builds an array of n bytes (or empty if n=0). All bytes, if any, are initialized to v (v is taken modulo 256). Raises Size if n<0.
`bytearray2 n`	Builds an uninitialized array of n bytes (or empty if n=0). Raises Size if n<0.
`length b`	Returns the number of bytes in bytearray b.
`sub (b,i)`	Returns component i of bytearray b. i ranges in 0..(length b - 1). Raises Subscript if i is out of range.
`update (b,i,v)`	Replaces component i in array b by v. Raises Subscript if i is out of range.
`bytearrayoflist l`	Builds a bytearray with its components taken in list l. Integers in list l are taken modulo 256.
`listofbytearray b`	Reverse of bytearrayoflist; makes a list from the components of a bytearray (in increasing index order).
`bytearrayofstring s`	Builds a bytearray with its component taken in string s.
`stringofbytearray b`	Reverse of bytearrayofstring; makes a string from the components of a bytearray, in increasing index order.

EXAMPLES

SEE ALSO

Dev

CONTENTS

lcs I/O devices

SIGNATURE

```
structure Dev =
   abstype dev
   exception Io : string -> exn
   val nulldev : dev
   val dopen : string * int -> dev
   val dclose : dev -> unit
   val dgetc : dev -> string
   val dgets : dev * int -> string
```

```
val dgetl : dev -> string
val dputs : dev * string -> unit
val dputl : dev * string -> unit
val flush : dev -> unit
val locks : dev * int -> bool
val is_term : dev -> bool
val de_dev : dev -> int * int * string
```

DESCRIPTION

`dev`
: Abstract type of lcs devices. Devices encode various structures to perform input/output, such as `instream`, `outstream`, `control`, `stream`, etc.

  This structure is still in experimental shape. It is intended as a system interface to stream I/O, for writing more elaborated structures from it (see e.g. `Io` or `Remote`). A device encapsulates flags, a buffer, a physical device descriptor, and ownership information.

`nulldev`
: The null device. Used at some places where a `dev` is required but not actually used.

`dopen (flags,n)`
: Opens a device. The flags tells what kind of device one want to open:

0	R	1	W
2	W append	3	W exclusive

`dclose d`
: Closes device d.

`dgetc d`
: Reads a character from device d.

`dgets (d,n)`
: Reads n characters from device d.

`dgetl d`
: Reads characters from `d` until end of line (transferred) or end-of-file (transferred) is read.

`dputs (d,s)`
: Writes string s on device d.

`dputl (d,s)`
: Writes string s, then an end-of-line character, on device d.

`flush d`
: Characters may be buffered in the lcs structures representing devices, and the actual writing delayed. flush forces writing all characters kept in the lcs device's buffer.

`locks (d,n)`
: Places or removes a lock on an lcs device; the scope of the lock is the lcs machine (not the host file system). This helps to prevent lcs processes running on the same lcs machine to perform concurrent I/O operations on a device. `locks` returns true if the operation succeeded. A device may be locked/unlocked on writing (W), on reading (R), or both (R+W), this is expressed by the second parameter of locks, as follows:

2	R unlock	3	R lock
4	W unlock	5	W lock
6	R+W unlock	7	R+W lock
_	no effect (`locks` returns false)		

`is_term d`
: Returns `true` if device `d` is of interactive kind, false otherwise.

`de_dev d`
: Returns a (rather crude) representation of the fields of a device. A better representation will be provided in a later release.

EXAMPLES

SEE ALSO

Prelude.Io, Network.Remote.

Evt

CONTENTS

Type of events.

SIGNATURE

```
structure Evt =
   event EXN : exn evt
```

DESCRIPTION

`'a evt`	Type of lcs events. The parameter is the type of messages brought. Like the type `exn` of exceptions, type `evt` is incrementally extended by specific declarations (**`event)`**. Like exceptions, events may be parameterized. To be useful, events must admit equality; and so must do their parameters, if any. On the other side, the type of messages they bring need not admit equality. These constraints are enforced by the type-checker. Both events parameters and the messages events carry are weakly polymorphic (like exceptions or references, events can't be created polymorphic).
EXN	A predefined event signaled by a process when an exception is raised within the process and is uncaught; the exception raised is passed as message of the event. Event EXN provides a means to observe the reasons of process failures. Contrarily to what happens in functional computations, an exception raised within a process does not propagate to the whole surrounding context, but has a scope restricted to that process. The EXN event extends the scope of the exception, but without its effects (an entrapped event just makes the issuing process terminate).

EXAMPLES

```
- event  E1
= and    E2 of int * int
= and    E3 with string
= and    E4 of int with bool;
event E1 : unit evt
event E2 : int * int -> unit evt
event E3 : string event
event E4 : int -> bool event

- fun A b c x = b catch
=        E1 => b
=     || E3 with "" => c | "end" => out! "end" => b
=     || E4 (4 * x) with m => if m then c else b;
val A = - : {out: string}_a -> {out: string}_a -> int -> {out: string}_a
```

SEE ALSO

Prelude.Exn.

Exn

CONTENTS

Type of exceptions.

SIGNATURE

```
structure Exn
```

DESCRIPTION

`'a evt`	Type of exceptions. Not expressible in terms of other types. Type exn is incrementally extended by **`exception`** declarations. Type `exn` does not admit equality. Exceptions are weakly polymorphic.

EXAMPLES

SEE ALSO

Prelude.Evt

Fun

CONTENTS

Type function and some frequently needed combinators.

SIGNATURE

```
structure Fun =
   abstype 'a -> 'b
   val I : 'a -> 'a
   val o : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
   val fun_pow : int -> ('a -> 'a) -> 'a -> 'a
   val repeat : int -> ('a -> 'b) -> 'a -> unit

   val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
   val pair : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
   val can : ('a -> 'b) -> 'a -> bool
   exception Assert : exn
   val assert : ('a -> bool) -> 'a -> 'a

   val fold : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
   val rev_fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
   exception Reduce : exn
   val reduce : ('a * 'a -> 'a) -> 'a list -> 'a
   val rscan : ('a * 'a -> 'a) -> 'a list -> 'a list
```

DESCRIPTION

`'a -> 'b`	Type of functions. Functions admit recursion; they do not admit equality.
`I`	identity function

`o`	function composition
`fun_pow n f x`	== f (f (.... (f x))); iterates n times application of f
`repeat n f x`	== (f x; f x; ...; f x); evaluates (f x) n times.
`curry f`	makes the paired function f a curried function
`pair f`	makes the curried function f a paired function
`can f x`	returns false if f x fails, and true otherwise
`assert p x`	returns x if (p x) true, raises exception Assert otherwise
`fold f [x1, ... ,xn] b`	`== f x1 (f x2 ( ... (f xn b)...)`
`rev_fold f b [x1, ... ,xn]`	`== (f ... (f (f b x1) x2) ...) xn)`
`reduce f [x1, ... ,xn]`	`== f (x1, f (x2,  ... xn ...))`
`rscan f [x1,x2,...,xn]`	`== x1 :: reduce f [x1,x2] ::  ... :: reduce f [x1,x2,...,xn]`

EXAMPLES

```
- fold (curry op +) [1,2,3,4,5] 0;
val it = 15 : int

- reduce (op +) [0,1,2,3,4,5];
val it = 15 : int

- rscan (op +) [0,1,2,3,4,5];
val it = [0, 1, 3, 6, 10, 15] : int list
```

SEE ALSO

Host

CONTENTS

Host operating system services.

SIGNATURE

```
structure Host =
   exception Host : string -> exn
   val date : unit -> int * int * int * int * int * int
   val get_env : string -> string
   val set_env : string -> unit
   val system : string -> unit
   val working_dir : unit -> string
   val change_dir : string -> unit
   val execute : string -> instream * outstream
```

DESCRIPTION

`date ()`	returns (sec,min,hour,day,month,year)
`system cmd`	sends host shell command `cmd` for execution

`working_dir ()`	returns working directory as a string
`change_dir` d	changes the working directory to `d`
`get_env v`	host execution environment access
`set_env c`	host execution environment update
`execute` cmd	execs host process `cmd` and links its stdin/stdout to lcs streams

Some of these functions may fail with Host and a string indicating the reason of failure if either the service requested is not supported by the host or evaluation terminated abnormally.

EXAMPLES

```
- get_env "EDITOR";
val it = "vi" : string

- set_env "EDITOR=emacs";
val it = () : unit

- change_dir "/tmp";
val it = () : unit

- working_dir ();
val it = "/tmp" : string
```

SEE ALSO

Int

CONTENTS

Signed fixed precision integers.

SIGNATURE

```
structure Int =
   datatype int   [int]
   exception Overflow : exn
   val ~ : int -> int
   val + : int * int -> int
   val - : int * int -> int
   val * : int * int -> int
   exception Div : exn
   val div : int * int -> int
   val mod : int * int -> int
   val divide : int * int -> int * int
   val < : int * int -> bool
   val > : int * int -> bool
   val <= : int * int -> bool
   val >= : int * int -> bool
   val abs : int -> int
   val min : int * int -> int
   val max : int * int -> int
   exception Pow : exn
```

```
val pow : int * int -> int
val maxint : int
val minint : int
val seed : unit -> unit
val rand : int -> int
exception Int_of_string : exn
val int_of_string : string -> int
val string_of_int : int -> string
val makestring : int -> string
val print : int -> int
```

DESCRIPTION

`int`	Type of finite precision integers. An int is an integer in the range `min-int..maxint`. Parser rejects out of range integers. All operations in this structure raise Overflow when computing an out of range integer.
`~, +, -, *, div, mod, divide`	The usual integer arithmetic primitives. ~ is inverse (unary -). All raise `Overflow` if the result is out of range. `div, mod, divide` raise `Div` when divisor is 0. `divide (x,y)` returns the pair (q,r) satisfying x = q*y + r, with either 0<=r<d, or d<r<=0;
`<, >, <=, >=`	`int` comparison functions.
abs x	Absolute value of x.
`min (x,y), max (x,y)`	Returns the min (resp. max) of x and y.
`pow (x,n)`	Returns x at power n.
`maxint, minint`	Constants holding the largest (resp. smallest) representable `int`.
`rand i`	returns a random integer in the interval `[0..i[`;
`seed()`	`i`nitializes the random generator.
`string_of_int n`	Returns the string representation of integer `n`.
`int_of_string s`	Returns the integer represented by string `s`. Raises `Int_of_string` if `s` does not represent an integer.
`makestring n`	Returns a string representing integer n.
`print n`	Identity function. Prints as a side effect the string representation of n.

EXAMPLES

```
- val x = 325;
val x = 325 : int

- divide (x,7);
val it = (46, 3) : int * int

- ~minint;
uncaught exception  --  Overflow
```

SEE ALSO

`Prelude.Real`: floating point arithmetics

`Prelude.Mp`:	multiple precision integer arithmetics
`Extras.Rat`:	fixed precision rational arithmetics
`Extras.Mprat`:	multiple precision rational arithmetics

Io

CONTENTS

Input and output streams, windows.

SIGNATURE

```
structure Io =
   exception Io : string -> exn

   (* instreams: *)
   abstype instream
   val std_in : instream
   val open_in : string -> instream
   val sopen_in : string -> instream
   val close_in : instream -> unit
   val getc : instream -> string
   val getl : instream * bytearray * int -> int
   val gets : instream * bytearray * int -> int
   val input : instream * int -> string
   val input_line : instream -> string
   val lookahead : instream -> string
   val end_of_stream : instream -> bool
   val is_term_in : instream -> bool
   val de_instream : instream -> dev

   (* outstreams: *)
   abstype outstream
   val std_out : outstream
   val open_out : string -> outstream
   val open_append : string -> outstream
   val open_excl : string -> outstream
   val sopen_append : string -> outstream
   val close_out : outstream -> unit
   val puts : outstream * string * int -> unit
   val output : outstream * string -> unit
   fun output_line : outstream * string -> unit
   val flush_out : outstream -> unit
   val is_term_out : outstream -> bool
   val de_outstream : outstream -> dev

   (* control streams: *)
   abstype control
   val std_ctrl : control
   val wait_ctrl : control -> string
   val de_control : control -> dev

   (* windows: *)
```

```
abstype window
val std_win : window
val open_set_win : string * string * (int * int) * (int * int) *
                   string * string * string * int
                   -> window
val open_win : unit -> window
val open_term : unit -> window
val close_win : window -> unit
val win_in : window -> instream
val win_out : window -> outstream
val win_ctrl : window -> control
val win_frame : window -> dev
val win_term : window -> string
val input_win : window * int -> string
val input_line_win : window -> string
val output_win : window * string -> unit
val output_line_win : window * string -> unit
val wait_ctrl_win : window -> string
val de_window : window -> control * instream * outstream * dev * string
```

DESCRIPTION

Datatypes `instream`, `outstream` and `control` are represented by devices (see structure Dev). Windows pack together an instream, an outstream, a control, a frame (a form of device), and a string (the terminfo string). Streams abstract host operating system files and interactive streams, windows abstract physical host windows.

`Io` — Exception Io is raised by a number of primitive or derived io operations to indicate exceptional conditions. The string parameter makes precise the reason of failure.

`Instreams`

`instream` — Abstract type of input streams. Does not admit equality.

`std_in` — The instream component of constant `std_win` (see below).

`open_in f` — Opens file `f` for reading, returns an `instream`.

`sopen_in f` — Same as `open_in`, except that it uses the lcs search path to locate file `f`.

`close_in ii` — Closes an `instream` for reading.

`getc ii, gets (ii,buff,n), getl (ii,buff,n)`

Raw input functions on instreams. All three functions raises `Io "eof"` if `ii` is closed or the end-of-file character is read.

`getc ii` reads a single character on `ii` and returns it.

`gets (ii,buff,n)` reads characters on `ii` and stores them in byte-array `buff` at offset `n`. It reads at most k =`length buff - n` characters, and returns the number of characters read.

`getl (ii,buff,n)` acts as `gets (ii,buff,n)` except that it returns as soon as a newline character is read.

`input (ii,n), input_line ii, lookahead ii`

Higher level input functions on instreams. All returns the string of characters read, or the empty string if `ii` is closed of the end-of-file character is read.

`input (ii,n)` reads `n` characters on `ii`.

`input_line ii` reads a line on `ii`, returned string includes the end-of-line character.

`lookahead ii` waits for a character to be available on `ii` and returns it, leaving it in the stream.

`end_of_stream ii` — Returns true is stream `ii` is closed.

`is_term_in ii` — Returns true if instream `ii` is interactive (created as a component of some window) or false otherwise (created by `open_in` call). Interactive instreams are closed by primitive `close_in`, or by typing in them the end-of-file character (Control D).

`de_instream ii` — Returns the lcs device representing instream `ii`.

`Outstreams`

`outstream` — Abstract type of output streams. Does not admit equality.

`std_out` — The outstream component of constant `std_win` (see below).

`open_out f` — Opens file `f` for writing, returns an outstream. If file f exists, then it is replaced by a new one, otherwise it is created.

`open_append f` — Opens an existing file `f` for writing, returns an outstream. If file f exists, then output to that outstream will be appended to the current content of f.

`open_excl f` — Opens file `f` for writing, returns an outstream. Fails if file f exists, othewise creates it.

`sopen_append f` — Save as `open_append f` except that it uses the lcs search path to locate file `f`.

`close_out oo` — Closes an `outstream` for writing.

`puts (oo,s,n)` — Raw output function. Writes on `outstream oo` the characters found in string `s` from offset `n`. Returns the number of characters written. Raises `Io "eof"` if `oo` is closed.

`output (oo,n), output_line oo`

Higher level output functions. Both raise `Io "puts"` if `oo` is closed.

`output (oo,s)` writes string `s` on `oo`.

`output_line (oo,s)` writes string `s`, followed by an end-of-line character, on `oo`.

`flush_out oo` — Functions `output` and `output_line` may delay actual writing of characters until some amount of them is available, or an end-of-line is to be written. `flush_out oo` forces writing of all characters buffered on the device abstracted by `oo`.

`is_term_out oo` — Returns true if outstream `oo` is interactive (created as a component of some window), or false otherwise (created by one of `open_out/ open_append/open_excl` calls).

`de_outstream oo` — Returns the lcs device representing outstream `oo`.

Controls

`control` — Abstract type of control streams. Does not admit equality. Control streams are used to transmit exceptional conditions in windows raised by typing some control characters in windows such as ^C (interrupt). Check the Language Reference Manuals fot the control sequences currently supported.

`std_ctrl` — The control component of constant `std_win` (see below).

`wait_ctrl cc` — Reads a control code on `control` stream `cc`. E.g. typing ^C in a window makes that character available on the control stream associated with that window; this has no other effect (it is up to the program that manages the window to take decisions according to the controls received).

`de_control cc` — Returns the lcs device representing control `cc`.

Windows

`window` — Abstract type of windows. Does not admit equality. A window encapsulate an instream, an outstream, a control, a frame, and a terminfo string.

`std_win` — Standard window constant identifying the window in which lcs is launched, set when entering lcs.

`open_set_win (tag,title,(xpos,ypos),(lins,cols),font,display,targs,lined)`

Opens a new window on the user's screen. These parameters are:

`tag` identifies the window too to be used to implement the window. Must be an entry in lcs file `bin/<target>/termspec` (see details on this file in installation procedure for lcs, in Release Notes).

`title` is the title given to the window.

`xpos,ypos` specify the position of window on screen (in pixels).

`lins,cols` specify the size of window, in lines and columns.

`font` is the font to be used in the window (there is a default font, associated with string "").

`display` is the display on which to create the window (there is a default display, associated with string "")..

`targs` are additional args passed to the window tool (these are not checked by lcs).

`lined` specifies the line discipline to be used in the window (none if 0, line discipline otherwise).

`open_win ()` — Opens a new window on the user's screen with default parameters. See implementation below.

`open_term t` — Same as open_win except specifies the window tool to be used (e.g. "xterm", or "cmdtool"). See implementation below. The tool name must be an entry in the lcs file `bin/<target>/termspec`.

`close_win w` — Closes window w, closes the physical window and all streams part of `w`.

`win_in w, win_out w, win_ctrl w, win_frame w, win_term w`

Return the components of window `w`, i.e. its instream, outstream, control, frame device, and terminfo string, respectively.

`input_win (w,n)`	Reads n characters on window w, eqv. `input (win_in w,n)`
`input_line_win`	Reads a line on window w, eqv. `input_line (win_in w)`
`output_win`	Writes string `s` on window w, eqv. `output (win_out w,s)`
`output_line_win`	Writes string `s` followed by n end-of-line character on window w, eqv. `output_line (win_out w,s)`
`wait_ctrl_win`	Reads a control character on window w, eqv. `wait_ctrl (win_ctrl w)`
`de_window`	Returns the components of a window as a tuple.

EXAMPLES

Opening windows, examples:

`open_set_win ("xterm","LCS",(100,50),(30,20),"","tippi:0.0","",1)`

Opens an xterm terminal window, with label "LCS", at position (100,50), with 24 lines and 80 columns, default font, on display tippi:0.0.

`open_set_win ("xterm","LCS",(100,50),(30,20),"","","-iconic -bg red",1)`

Same tool, title, position and size, but with the default font, display, a red background, started iconized. The additional arguments passed to the tool are NOT checked by lcs.

`open_set_win ("cmdtool","LCS",(100,50),(24,80),"","",1)`

Opens a sun-openwindow's cmdtool, using default font and display.

`open_set_win ("","LCS",(100,50),(24,80),"","",0)`

Opens a terminal window of default kind, and with raw line discipline (characters are available as soon as typed, without interpretation). There is a "default" entry in lcs file `bin/<target>/termspec` to which the default windowtool is bound; please update that file if the paths or tools in it are uncorrect at your site;

`open_win ()`

Opens a window of kind "default", with default parameters.

`open_term "cmdtool";`

Same as open_win except that opens a "cmdtool" window instead of default tool.

SEE ALSO

Prelude.Stdio: specialized input/output functions for top-level window.

Lift

CONTENTS

Type lift of lazy values.

SIGNATURE

```
structure Lift =
   type lift
   con up   : 'a -> 'a lift;
   val down : 'a lift -> 'a;
```

DESCRIPTION

`'a lift`	Type of delayed evaluation (or lazy) values. Admits recursion. Does not admit equality.
`up e`	Builds a lazy value encapsulating `e`. Constructor `up` is non strict, that is expression `e` in `up e` is not evaluated, though (`up e`) is a value. Evaluation of the content of a lazy value is obtained by applying function `down` to it, or through extraction of its content by pattern matching.
`down v`	Value `v` is a lazy value. Evaluates its body and returns it. As a side-effect, updates all occurrences of value **v** in the environment with its evaluated content.

EXAMPLES

```
- val x = up 12
- val y = x;
val x = up - : int lift                              (body of x is not evaluated)
val y = up - : int lift                              (x and y denote the same value)

- down x;
val it = 12 : ift                                    (body of x now evaluated)

- (x,y);
val it = (up 12, up 12) : int lift * int lift        (body of y is evaluated too)

- fun ite (x,up y,up z) = if x then y else z;        (a functional conditional)
val ite = - : bool * 'a lift * 'a lift -> 'a

- fun fact n = ite (n=0,up 1,up (n*fact (n-1)));
val fact = - : int -> int
```

SEE ALSO

Extras.Lizt: lazy lists

List

CONTENTS

Lists.

SIGNATURE

```
structure List =
   type 'a list
   con nil : 'a list
   con :: : 'a * 'a list -> 'a list
   exception Hd : exn
   val hd : 'a list -> 'a
```

```
exception Tl : exn
val tl : 'a list -> 'a list
val null : 'a list -> bool
val length : 'a list -> int
val @ : 'a list * 'a list -> 'a list
val mem : 'a -> 'a list -> bool
val rev : 'a list -> 'a list
val flat : 'a list list -> 'a list
val map : ('a -> 'b) -> 'a list -> 'b list
val filter : ('a -> bool) -> 'a list -> 'a list
exception Find : exn
val find : ('a -> bool) -> 'a list -> 'a
val split : ('a * 'b) list -> 'a list * 'b list
exception Combine : exn
val combine : 'a list * 'b list -> ('a * 'b) list
val string_of_list : string * string * string
              -> ('a -> string) -> 'a list -> string
```

DESCRIPTION

`'a list` — Type of polymorphic homogeneous lists. Type `ty list` admits equality if and only if argument type `ty` does.

`[], nil` — Empty list constructor.

`e :: l` — Constructor `::` is the "cons" constructor for lists. `e :: l` is the list obtained by prefixing list `l` with element `e`. Notation $[e_1, e_2, \ldots, e_n]$ is a shorthand for $e_1 :: e_2 :: \ldots :: e_n :: nil$.

`hd l, tl l, null l, length l`

Basic list primitives:

`hd l` returns the first element of list `l`, raises `Hd` if `l` is empty.

`tl l` returns `l` without its first element, raises `Tl` if `l` is empty.

`null l` returns `true` if list `l` is empty, or `false` otherwise.

`length l` returns the number of elements in `l`.

l1 @ l2 — Returns the concatenation of lists `l1` and `l2`.

mem e l — Returns `true` if element e belongs to list `l`, `false` otherwise.

rev l — Returns the reverses (transpose) of list l.

flat l — `l` is a list of lists, concatenates all its list elements.

map f l — Returns a list in which the i^{th} element is obtained by applying function f to the i^{th} element of `l`.

filter p l — Returns the list constituted of the elements of `l` satisfying predicate `p`.

find p l — Returns the first element of `l` satisfying predicate p. Raises `Find` if none is found.

split l — List l is a list of pairs. Returns a pair of list (l1,l2) in which l1 holds the first components of these pairs, and l2 holds their second components.

combine l — Inverse of split. Builds a list of pair from a pair of lists.

```
string_of_list (b,s,e) p l
```

Utility for building string representations of lists; makes a string out of list `l`. `b`, `s`, and `e` are the list beginning bracket, the element separator, and the list end bracket, respectively (all are strings). Function `p` makes a string out of an element of list `l`.

EXAMPLES

SEE ALSO

`Extras.Alist`: associative lists
`Extras.Slist`: sets as lists
`Extras.Tlist`: arrays as lists
`Extras.Lizt`: lazy lists

Mp

CONTENTS

Multiple precision integers.

SIGNATURE

```
structure Mp =
   datatype mp   [mp]
   val ~ : mp -> mp
   val + : mp * mp -> mp
   val - : mp * mp -> mp
   val * : mp * mp -> mp
   exception Div : exn
   val div : mp * mp -> mp
   val mod : mp * mp -> mp
   val divide : mp * mp -> mp * mp
   val < : mp * mp -> bool
   val > : mp * mp -> bool
   val <= : mp * mp -> bool
   val >= : mp * mp -> bool
   val abs : mp -> mp
   val min : mp * mp -> mp
   val max : mp * mp -> mp
   val pow : mp * mp -> mp
   val mp : int -> mp
   exception Cut : exn
   val cut : mp -> int
   val string_of_mp : mp -> string
   exception Mp_of_string : exn
   val mp_of_string : string -> mp
   val print : mp -> mp
   val makestring : mp -> string
```

DESCRIPTION

`mp` — Type of arbitrary precision integers. `mp` integer constants are represented prefixed by digit 0. Type `mp` admits equality.

`~, +, -, *, div, mod, divide`	The usual integer arithmetic primitives. ~ is negation (unary -). `div, mod, divide` raise `Div` when divisor is 0. `divide (x,y)` returns the pair (q,r) satisfying x = q*y + r, with either 0<=r<d, or d<r<=0;
`<, >, <=, >=`	`mp` comparison functions.
abs x	Absolute value of x.
`min (x,y), max (x,y)`	Returns the min (resp. max) of x and y.
`pow (x,n)`	Returns x at power n.
`mp i, cut m`	`int` to `mp`, and `mp` to `int` conversions, respectively. `cut m` raises `Cut` if `m` larger than (`mp maxint`) or smaller than (`mp minint`).
`string_of_mp m`	Returns the string representation of `mp` integer `m`.
`mp_of_string s`	Returns the `mp` integer represented by string `s`. Raises `Mp_of_string` if `s` does not represent a `mp` integer.
`makestring m`	Returns a string representing `mp` integer `m`.
`print m`	Identity function. Prints as a side effect the string representation of `m`.

EXAMPLES

```
- (0,00);
val it = (0, 00) : int * mp

- 02 pow 0256;
val it = 0115792089237316195423570985008687907853269984665640564039457
7584007913129639936 : mp
```

SEE ALSO

`Prelude.Int, Prelude.Real, Extras.Rat, Extras.Mprat.`

Object

CONTENTS

Machine representation of lcs values.

SIGNATURE

```
structure Object =
   abstype object
   exception Obj : exn
   val is_boxed_obj : object -> bool
   val is_loc_obj : object -> bool
   val is_vect_obj : object -> bool
   val eq_obj : object * object -> bool
   val pack_obj_list : int list -> object
   val mk_obj : 'a -> object
   val dest_obj : object -> 'a
   val cast : 'a -> 'b
   val pack_obj : object -> string
```

```
    val unpack_obj : string -> object
    val clone : 'a -> 'a
    val exece : object Vector.vector * object -> object
```

DESCRIPTION

`object`	Abstract type of representations of lcs values. Type `object` does not admit equality. Objects interface the LCS virtual machine with the compiler and printer. The functions below flagged with UNSAFE are type-unsafe, they may crash the lcs virtual machine if the assumptions made are violated.
`is_boxed_obj ob`	Returns true if `ob` is boxed.
`is_loc_obj ob`	Returns true if `ob` is mutable.
`is_vect_obj ob`	Returns true if `ob` is a vector cell.
`eq_obj (o1,o2)`	UNSAFE. Objects o1 and o2 are assumed to represent constants of the same lcs type. Returns true if o1 and o2 are equal (in the sense of equality of constants).
`pack_obj_list l`	Packs a list `l` of bytes into a byte vector object.
`mk_obj e`	Makes an object out of any lcs value e.
`dest_obj ob : ty`	UNSAFE. Object ob is assumed to represent a value of type ty; makes a value of type ty out of object ob.
`cast e : ty`	UNSAFE. Combines `mk_obj` and `dest_obj`. Assumes `e` has the representation of an object of type `ty;` makes `e` a value of type `ty`.
`pack_obj` ob	Packs object `ob` into a string.
`unpack_obj s`	Inverse of `pack_obj`, unpacks a string into an object.
`clone e`	Returns a clone of value e, preserving its type. A clone is a value with same structure, but not sharing any mutable cells with the original. eqv. `dest_obj (unpack_obj (pack_obj (mk_obj (e : 'a)))) : 'a`
`exece` (E,ob)	UNSAFE. Assumes `ob` represents some code object. Evaluates code `ob` in environment `E`, returns the result of evaluation as an `object`.

EXAMPLES

SEE ALSO

Pair

CONTENTS

Pairs.

SIGNATURE

```
structure Pair =
    val fst : 'a * 'b -> 'a
    val snd : 'a * 'b -> 'b
```

DESCRIPTION

`*`	An instance of the product type of SML. Two pairs are equal if their corresponding components are equal.
`fst p, snd p`	Return the left and right components of pair `p`, respectively.

EXAMPLES

SEE ALSO

Prelude

CONTENTS

Initial bindings.

SIGNATURE

```
structure Prelude =
   structure Dev = ...
   structure Io = ...
   structure Stdio = ...
   structure Exn = ...
   structure Unit = ...
   structure Pair = ...
   structure Bool = ...
   structure Int = ...
   structure Real = ...
   structure Mp = ...
   structure String = ...
   structure Ref = ...
   structure Lift = ...
   structure List = ...
   structure Vector = ...
   structure Array = ...
   structure Bytearray = ...
   structure Fun = ...
   structure System = ...
   structure Host = ...
   structure Object = ...
   (* next three omitted if process option is -n *)
   structure Evt = ...
   structure Void = ...
   structure Beh = ...
```

DESCRIPTION

This structure contains all structures defined in this section (except itself). It is initially loaded and installed, it can be reloaded and reinstalled by reading in script `lprelude` (see Section 3).

EXAMPLES

SEE ALSO

Real

CONTENTS

Real numbers.

SIGNATURE

```
structure Real =
   datatype real   [real]
   exception Real : string -> exn
   val ~ : real -> real
   val + : real * real -> real
   val - : real * real -> real
   val * : real * real -> real
   exception Div : exn
   val / : real * real -> real
   val < : real * real -> bool
   val > : real * real -> bool
   val <= : real * real -> bool
   val >= : real * real -> bool
   val abs : real -> real
   val min : real * real -> real
   val max : real * real -> real
   val real : int -> real
   val floor : real -> int
   val round : real -> int
   val truncate : real -> int
   exception Mathlib : string -> exn
   val mathlib : int * real -> real
   val sin : real -> real
   val cos : real -> real
   val arctan : real -> real
   val pow : real * real -> real
   exception Sqrt : exn
   val sqrt : real -> real
   exception Exp : exn
   val exp : real -> real
   exception Ln : exn
   val ln : real -> real
   val pi : real
   val e : real
   val rrand : unit -> real
   val string_of_real : real -> string
   exception Real_of_string : exn
   val real_of_string : string -> real
   val makestring : real -> string
   val print : real -> real
```

DESCRIPTION

`real`	Type of floating point numbers. Parser rejects reals that do not fit in machine representation. Type real admits equality.
`~, +, -, *, /`	The usual integer arithmetic primitives. ~ is inverse (unary −). `/` raises `Real "division"` when dividing a nonnull real by `0.0`, and `Real`

"invalid" when dividing 0.0 by 0.0. Other Real exceptions possibly raised by these primitives are Real "overflow" and Real "underflow".

`<, >, <=, >=`
: real comparison functions.

`abs r`
: Absolute value of r.

`min (x,y), max (x,y)`
: Returns the min (resp. max) of x and y.

`real i`
: int to real conversion.

`floor r, round r, truncate r`
: real to int conversions. All three raise Floor if result of conversion is not in range Int.minint..Int.maxint.

  floor r returns the largest integer smaller than r.

  round r returns the integer nearest to r.

  truncate r returns the mantissa of r.

`mathlib (n,r)`
: Provides a number of unary math functions, according to the first component of its argument:

1 sin	2 cos	3 tan	4 asin	5 acos	6 atan
7 sinh	8 cosh	9 tanh	10 asinh	11 acosh	12 atanh
13 sqrt	14 cbrt	15 ln	16 exp		

  Primitive mathlib raises Mathlib when first argument is not in range 1..16. In addition, some functions obtained from mathlib may raise exception Real on some conditions, with a string indicating the nature of the exception.

`sin, cos, arctan, sqrt, exp, ln`
: Implement from mathlib the obvious mathematical functions. Raise specific exceptions (rather than Real or Mathlib).

`r pow r'`
: Raises r to power r'.

`pi, e`
: The mathematical constants.

`rrand ()`
: Random number generator. Returns a random real in the interval]0.0, 1.0[.

`string_of_real r`
: Returns the string representation of real r.

`real_of_string`
: Returns the real represented by string s. Raises Real_of_string if s does not represent a real.

`makestring r`
: Returns a string representing real r.

`print r`
: Identity function. Prints as a side effect the string representation of r.

EXAMPLES

SEE ALSO

Ref

CONTENTS

Value references.

SIGNATURE

```
Structure Ref =
   datatype 'a ref   ['a ref]
   con ref : '1a -> '1a ref
   val ! : 'a ref -> 'a
   val := : 'a ref * 'a -> unit
   val inc : int ref -> unit
   val dec : int ref -> unit
   val tns : bool ref -> bool
```

DESCRIPTION

`'a ref`	Type of references, or pointers. Type `ref` always admits equality; for references, equality means identity. Reference values are weakly polymorphic (as shown by the type of constructor `ref`). However, constructor `ref` may appear in patterns with type `'a -> 'a ref.`
`ref e`	Builds a reference holding the value of `e`.
`! r`	Returns the content of reference `r`.
`r := e`	Replaces the content of `r` with the value of `e`.
`inc r`	eqv. `r := !r + 1`
`dec r`	eqv. `r := !r - 1`
`tns r`	Test and set primitive. `r` is a reference to a `bool;` if `r` holds `true`, then returns `false`, otherwise turns `r true` and returns `true`. This is performed atomically, that is, if several lcs processes perform (`tns r`) simultaneously, then only one will eventually succeed. `tns` is useful for setting up locks in concurrent programs.

EXAMPLES

SEE ALSO

Stdio

CONTENTS

Input and output on top-level default window.

SIGNATURE

```
structure Stdio =
   val std_window : unit -> window
   val redirect_window : window -> window

   val std_input : unit -> instream
   val read : int -> string
   val read_line : unit -> string
```

```
val redirect_input : instream -> instream

val std_output : unit -> outstream
val write : string -> unit
val write_line : string -> unit
val redirect_output : outstream -> outstream

val std_control : unit -> control
val wait : init -> string
val redirect_control : control -> control
```

DESCRIPTION

The lcs toplevel operates on a default window, initially holding `std_win`. These primitives allow one to access and update the standard window, and provide IO primitives specialized for that window.

`std_window ()`	Returns the toplevel default operating window, conventionally called the "standard window".
`redirect_window w`	Installs w as the toplevel's standard window, returns the previous standard window. With process option "-n", `redirect_window` has no effect on Interrupt handling (in that interface, keyboard interrupt is only possible from `std_win`). With the "-l" (default) or "-s" option, this redirects sensitivity to keyboard interrupts to window `w`.
`std_input ()`	The standard window's instream. eqv. `win_in (std_window ()).`
`read n`	Reads n characters on the standard window. eqv. `input_win (std_window (), n).`
`read_line ()`	Reads a line on the standard window. eqv. `input_line_win (std_window ()).`
`redirect_input ii`	Replaces the input stream of the standard window by `ii`, returns the previous input stream.
`std_output ()`	The standard window's outstream. eqv. `win_out (std_window ()).`
`write s`	Writes string s on the standard window. eqv. `output_win (std_window (), s).`
`write_line s`	Writes string s, followed by an end-of-line character, on the standard window. eqv. `output_line_win (std_window (), s).`
`redirect_output oo`	Replaces the output stream of the standard window by `oo`, returns the previous output stream.
`std_control`	The standard window's control stream. eqv. `win_ctrl (std_window ()).`
`wait`	Expects a control character on the standard window control stream. eqv. wait_ctrl_win `(std_window ()).`
`redirect_control cc`	Replaces the control stream of the standard window by `cc`, returns the previous control stream. As for `redirect_window`, `redirect_control` has no effect on keyboard interrupts with process option "-n". With the "-l" or "-s" option, it redirects sensitivity to keyboard interrupts to the window the control stream `cc` belongs to.

EXAMPLES

```
- val oo = redirect_output (open_out "logfile");
val oo = - : outstream

- write "hello";
(* output is now printed in file logfile *)
- close_out (redirect_output oo);
(* recovering initial standard output, and closing the former outstream *)

- 1;
val it = 1 : unit
```

SEE ALSO

String

CONTENTS

Strings.

SIGNATURE

```
structure String =
   datatype string   [string]
   exception Ord : exn
   val ord : string -> int
   exception Chr : exn
   val chr : int -> string
   exception Size : exn
   val size : string -> int
   val string_length : string -> int
   val < : string * string -> bool
   val <= : string * string -> bool
   val > : string * string -> bool
   val >= : string * string -> bool
   exception Substring : exn
   val substring : string * int * int -> string
   val ^ : string * string -> string
   val implode : string list -> string
   val implode_sep : string -> string list -> string
   val explode : string -> string list
   val explode_sep : string -> string -> string list
   val makestring : string -> string
   val print : string -> string
```

DESCRIPTION

`string`	Type of `strings`, or character vectors. Strings admit equality. strings are represented between """ quotes; escape sequences are documented in Appendix 1 of the *Language Reference* Manual.
`ord s`	Returns ascii code for the character in string `s`. Raises `Ord` if s empty or holds more than 1 character.

`chr n`	Returns the string constituted of the character of ascii code `n`. Raises `Chr` if `n` out of range.
`size s, string_length s`	Returns number of characters in string `s`.
`<, <=, >, >=`	Lexicographic comparison functions for strings.
`substring (s,len,off)`	Returns the substring of `s` of length `len` starting at offset `off`. Raises `Substring` if not possible.
`s1 ^ s2`	Concatenates strings s1 and s2.
`implode l`	Concatenates the strings found in list `l`.
`explode s`	Explodes string s into a list of one-character strings.
`implode_sep sep l`	Concatenates the strings found in list `l`, adding separator `sep` between them.
`explode_sep sep s`	Takes `sep` as a separator. Returns the list of substrings of `s` separated by `sep` (or the list `[s]` if `sep` does not occur in `s`).
`makestring s`	Identity function on strings.
`print s`	Identity function, prints string s as a side-effect.

EXAMPLES

SEE ALSO

System

CONTENTS

```
System functions.
```

SIGNATURE

```
structure System =

   exception Interrupt : exn
   exception Bind : exn
   exception Match : exn
   datatype 'a option  [''a option]
   con None : 'a option
   con Some : 'a -> 'a option

   val sleep : real -> unit

   val protect : int -> int

   val init_path : unit -> string list
   val search_path : unit -> string list
   val add_path : string -> string list
   val rem_path : string -> string list
   val add_lib_path : string -> string list

   val read_file : string * bool * bool * bool -> unit
```

```
val use : string -> unit
val needs : string -> unit
val incl : string -> unit

exception Dump : string -> exn
exception Rest : string -> exn
val dump : string * string -> unit
val restore : string -> unit
val save : string -> unit
val load : string -> unit

val clean : unit -> unit
val exit : unit -> unit
val quit : unit -> unit

val lcs_pretty_printing : bool -> unit
val lcs_print_depth : int -> unit
val lcs_print_init : unit -> unit
val lcs_print_width : int -> unit

val argv : string Vector.vector
val arg_list : unit -> string list
val version : unit -> string * real * int
val runtime : unit -> int * int
val stats : unit -> int * int * int * int * int

val time : ('a -> 'b) -> 'a -> 'b
val timer : bool -> bool
val timers : bool -> bool
val print_code : bool -> bool

val gc : unit -> unit
val pp : unit -> unit

val prompt : string * string -> unit
val balance : unit -> unit
val depth : unit -> int list
val extended_charset : bool -> unit
val factorize_types : bool -> unit
val balance_types : bool -> unit
val refutable_lazy_patts : bool -> unit
```

DESCRIPTION

System exceptions:

`Interrupt`	Raised when interrupt key hit (with process option -n).
Bind	Raised when pattern matching fails in a binding.
Match	Raised for other pattern matching failures.

Option type:

`'a option`	Option type. Convenient to encode optional values.
`None, Some v`	Option type constructions.

Sleep primitive:

sleep t	Returns after `t` seconds elapsed. May be used concurrently in several lcs processes.

Raise protection level:

`protect n`	Sets protection level for evaluation of current process to `max (n, current level)`. Current level is initially defined from the `-p` flag of `lcs` command. Returns new protection level for current process. Does not affect the protection level of other lcs processes running on the same lcs machine.

Search path manipulation:

Functions `Io.sopen_in, Io.sopen_append`, as well as `read_file` and `load` below, use a search path for locating host files.

`init_path ()`	Initializes search path from info captured by lcs command (see the Release Notes for installation of lcs command).
`search_path ()`	Returns current search path.
`add_path p`	Adds path p to search path.
`add_lib_path p`	Same, p is relative to the lcs libraries directory.
`rem_path p`	Removes p from search path.

Reading in lcs commands from files (these functions should be used at top level only):

`read_file (f,e,i,c)` Reads the content of file f at lcs top level. The arguments are:

- f: file argument. Reads the first file found in the sequence: f.lcs (except if process option -n); f.sml, and f. At each attempt, the current search path is used;
- e: echo flag. If true, then the content of the file is printed on the standard window while read in;
- i: interactive flag. If true, then the bindings realized are printed on the standard window, otherwise, a simple period is printed for every command successfully read;
- c: conditional/imperative read flag. If true, then the file is read only if it was not read yet in the same lcs session (beware: read_file does not check for file changes nor for rebinding of yet loaded functions);

`use` f	Imperative read, prints bindings. eqv. `read_file (f,false,true,false)`
`needs f`	Conditional read, does not print bindings. eqv. `read_file (f,false,false,true)`
`incl` f	Imperative read, echoes commands and prints bindings. eqv. `read_file (f,true,true,false)`

Clean, quit, etc:

clean ()	Closes all currently open lcs devices (streams, windows, etc).
exit ()	Terminates lcs.
quit ()	Better exit of lcs: Redirects input/output on the standard window, then exits.

Dump, restore, etc:

`dump (app,p)`	Creates a portable core image with name `app` and default command parameters `p`. Raises `Dump` with an indicating string message if some condition prevents dump to succeed. Application `app` enters at the point following execution of `dump` (see examples of use in demos).
`save` app	Same as dump, except creates an application with default lcs parameters, and initializes the search_path on reentry and prints the user interface banner. Typically used to save interactive sessions for later use.
`restore app`	Restores core image `app`. The code captured in f replaces the current code. May Raise exception `Rest` with a string argument indicating the cause of the failure.
`load` app	Same as restore, except that uses the lcs search path to locate the application loaded.

Lcs pretty-printer controls:

`lcs_pretty_printing b`	Enables (if b true) or disables pretty-printing of command results (default enabled).
`lcs_print_init` ()	Reinitializes the pretty-printer (in case it has been confused).
`lcs_print_depth` d	Sets pretty-printing depth to d (useful for printing e.g. lon lists or arrays).
`lcs_print_width w`	Set page width assumed by printer to w characters.

Lcs configutation information, arg list, etc:

`argv` — Unix command line, as a `string vector`, with components:

0	runtime file called
1	bytecode file loaded
2,3	heap sizing parameters
4,5	garbage collection parameters
6	verbosity parameter
7	timeslice parameter
8	the bin directory
9	the share directory
10	the lib directory
11	the etc directory
12	arguments for the application
13	the lcsd server host
14	the lcsd server port number

`arg_list` ()	Returns the list of parameters passed to lcs by the command line (see Release notes or the lcs man page for how to pass parameters to an lcs applications).
`version ()`	Returns version information for interface in use.
`runtime ()`	Returns run and gc times (in ms) from beginning of session, times include user and system times.
`stats ()`	Statistics about the current lcs session. Returns:

- the number of minor garbage collections;
- the number of major garbage collections;

- the total amount of space allocated (in kilobytes);
- the number of process prunings;
- the total number of processes created.

Timing and profiling facilities:

`time` f x	Evaluates `f x,` prints in addition run and gc times, plus space allocated and gc count (minor and major).
`timer b`	For each command, prints time, gc and space information as above. Information is only printed for the evaluation pass. With process option -l or -s, and when the command is either **`call`** or **`lcall`**, prints in addition the number of processes created and process pruning information.
`timers b`	Same as `timer`, except that prints in addition time, space and gc information for all other user interface passes.
`print_code b`	Prints code executed for each toplevel command (if the user interface was generated with the DEBUG flag set).

Miscellaneous:

`prompt` (p,s)	Sets primary and secondary prompt (initially `"\n- "` and `"= "`).
`gc ()`	Forces a garbage collection and prints storage information.
`pp` ()	Forces a process pruning and prints process information (no effect if process option is -n)
`depth(), balance()`	Returns depths of values and tokens environments, and balances these (they are encoded as search trees). Warning: This does not necessarily improve the compiler performances.
`extended_charset b`	Makes parser, printer, and `char` primitive accept 128 character set (if false) or the 256 character set (if true).
`factorize_types b`	Controls factorization of behavior types (default true, no effect if -n).
`balance_types b`	Controls balancing of behavior types (default false, no effect if -n).
`refutable_lazy_patts b`	Enables or disables refutable lazy patterns (see Language Reference Manual for details).

EXAMPLES

With process "-l" or "-s":

```
(* ones define the toplevel of an application as a behavior: *)
- fun top args = ...;
val top = - : _a

(* then generates a standalone application named "app", not printing the lcs banner when called
   (sticky parameters are set to "-q"), and running top (arg_list ()) by: *)
- reboot do (dump ("app","-q")) => top (arg_list ());
```

With option "-n":

(* A function `exportFn`, emulating the njsml `exportFn` function, can be defined as follows (not with process option "-l" or "-s", as **`reboot`** is a reserved word in these interfaces): *)

```
fun exportFn (s,f) = (* emulates nj-sml exportFn *)
```

```
    reboot (fn () => (dump (s,""); f (arg_list ())));
```

SEE ALSO

Unit

CONTENTS

Empty product type.

SIGNATURE

```
structure Unit =
   datatype unit [unit]
   con () : unit
   val string_of_unit : unit -> string
   exception Unit_of_string;
   val unit_of_string : string -> unit
   val print : unit -> unit
   val makestring : unit -> string
```

DESCRIPTION

`unit`	A one-element constant type. Type unit is often used as "dummy argument" type or "dummy result" type. Admits equality.
`()`	Value "Empty", single value of type `unit`. Allowed in patterns.
`string_of_unit v`	String representation of a value v of type unit.
`unit_of_string` s	Returns the value of type unit represented by string s. Raises exception `Unit_of_string` if s does not represent a value of type `unit`.
`print`	Identity function; prints representation of Empty.
`makestring`	Makes a string from a value of type unit.

EXAMPLES

SEE ALSO

Vector

CONTENTS

Vectors.

SIGNATURE

```
structure Vector =
   datatype 'a vector   [''a vector]
   val vector : 'a list -> 'a vector
   val length : 'a vector -> int
   exception Subscript : exn
   val sub : 'a vector * int -> 'a
   exception Size : exn
```

```
val tabulate : int * (int -> 'a) -> 'a vector
```

DESCRIPTION

`'a vector`	Type of vectors, or homogeneous products. Their component can be accessed by indexing. Vectors admit equality iff their components do. Two vectors are equal if they have same length and have equal contents at all indices.
`#[], #[v1,...]`	Vectors are represented like lists, but preceded by symbol “#”. Vectors are not allowed in patterns.
`vector l`	Produces a vector from the list l of its components.
`length v`	Returns number of components of v.
`sub (v,i)`	Returns component at index i in v; indices range from 0 to (length v - 1). Raises exception Subscript if index out of range.
`tabulate (n,f)`	Produces a vector of length n with (f i) as i^{th} component, for i in 0..n-1. Raises exception Size if size parameter is negative.

EXAMPLES

```
- open Vector;
- #[];
val it = #[] : 'a vector
- vector [1,2,3,4,5];
val it = #[1, 2, 3, 4, 5] : int vector
- sub (it,1);
val it = 2 : int
```

SEE ALSO

`Prelude.String` (similar to vectors of characters)
`Prelude.Array`
`Prelude.Bytearray`

Void

CONTENTS

Types `void` and monomorphic behavior type constant “_”.

SIGNATURE

```
structure Void =
   abstype void
   abstype _
```

DESCRIPTION

`void`	`void` is a “dummy” type used to built the monomorphic behavior type _. Type `void` does not admit equality.
`_`	`Type _` (read void behavior type) is a behavior type in which every label is ascribed an extension and a message of type `void`. Since `void` is a monotype, _ is too. Types _ is used through type constraints to force the type of a behavior to be monomorphic, which makes possible to built references out of them (see example below), or to built exceptions encap-

sulating behaviors, or to pass behaviors over streams between lcs sites (see Remote). As all behavior types, type _ does not admit equality.

EXAMPLES

```
- val r = ref (p!12 => stop : {p:int}_);
val r = ref - : {p: int}_ ref

- agent b = p? _ => stop /\ !r;
val b = - : {p: int}_
```

SEE ALSO

2.3 Extras: general purpose additional structures

This section describes structure `Extras` and all of its sub-structures. Structure `Extras` is not initially loaded; it can be loaded and installed by reading in script `lextras` (see Section 3). Its definition is found in the sequel, followed by the definitions of all of its sub-structures.

Alist

CONTENTS

Lists accessed by contents, and association lists.

SIGNATURE

```
structure Alist =
   exception Assoc : exn
   val assoc : ''a -> (''a * 'b) list -> 'b
   exception Rev_assoc : exn
   val rev_assoc : ''a -> ('b * ''a) list -> 'b
   exception Chop_find : exn
   val chop_find : ('a -> bool) -> 'a list -> 'a list * 'a list
   val drop : ''a -> ''a list -> ''a list
   exception Replace : exn
   val replace : ''a -> ''a -> ''a list -> ''a list
```

DESCRIPTION

`assoc k l`	List l is taken as a list of (key,content) pairs. Returns content associated with key k in l. Raises Assoc if key k not found in fst of any component of l.
Rev_assoc k l	As Assoc, except that keys and contents are transposed. l holds pairs (content,key).
`chop_find p l`	Returns a pair of lists (l1,l2) such that l1@l2=l and the hd of l2 is the first element of l satisfying predicate p. Raises `Chop_find` if no element of l satisfies p.
`drop e l`	Returns l in which the first occurrence of element e is removed.
`replace x y l`	Returns l in which the first occurrence of element x is replaced by v. Raises `Replace` if element v not found in l.

EXAMPLES

SEE ALSO

Avltree

CONTENTS

AVL trees.

SIGNATURE

```
structure Avltree =
   abstype 'a stree
   val nullstree : 'a stree
```

```
exception Lookupstree : exn
val lookupstree : 'a stree -> string -> 'a
val updatestree : 'a stree * (string * 'a) -> 'a stree
val streedepth : 'a stree -> int
val balancestree : 'a -> 'a
val list_of_stree : 'a stree -> (string * 'a) list
```

DESCRIPTION

`'a stree`	Abstract type of AVL trees with string keys. Interface is compatible to that of structure Stree, which implements a type of binary search trees. Type `stree` does not admit equality.
`nullstree`	Returns an empty AVL tree.
`lookupstree` t k	Returns the item associated with key k in tree t. Raises `Lookupstree` if none is found.
`updatestree t (k,v)`	If some item is stored under key k in tree t, then replaces the item by v. Otherwise the entry (k,v) is added to the tree. AVL trees are built from `nullstree` and `updatestree`.
`streedepth t`	Returns the depth of the deepest item in tree t.
`balancestree t`	Identity function. Provided for compatibility with structure Stree. AVL trees are balanced incrementally.
`list_of_stree t`	Builds a list from all pairs (key,item) stored in tree t, in increasing lexicographic order of keys.

EXAMPLES

SEE ALSO

Bench

CONTENTS

Some arithmetic functions often used for tests or quick estimation of performances.

SIGNATURE

```
structure Bench =
   val fact : int -> int
   val fact : mp -> mp
   val fib : int -> int
   val fib : mp -> mp
   val ftak : int -> int -> int -> int
   val ptak : int * int * int -> int
   val ackermann : int -> int -> int
```

DESCRIPTION

`fact n`	Factorial of `n`, overloaded for `int` and `mp` integers.
`fib n`	Computes sum of fibonacci series with naive algorithm (and fib 0 = 1), overloaded for `int` and `mp` integers.
`ptak(x,y,z), ftak x y z`	Takeuki function, product and curried versions.

ackermann n m — Ackermann function.

EXAMPLES

```
- fact 0100;
val it =
0933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000 :
mp

- timer true;
val it = true : bool

- fib 30;
3.06s 0.00s 0kb (0+0)gc
val it = 1346269 : int

- ftak 18 12 6;
0.10s 0.00s 1kb (0+0)gc
val it = 7 : int
```

SEE ALSO

Broadcast

CONTENTS

Broadcasting communication medium abstraction.

SIGNATURE

```
structure Broadcast =
   val broadcast  : int -> {inp.int: 'a, out.int: 'a}_b
   val broadcast2 : int -> {inp: int * 'a, out.int: 'a}_b
```

DESCRIPTION

`broadcast n` — Abstract a communication medium to be used to interconnect a number `n` of processes. A value read on port `inp#i`, with `i` in `1..n`, is offered on all ports `out#k`, with k in `1..n` -{i}. The medium simulates an atomic broadcast in the sense that it waits for all output offers to be consumed before accepting another input.

`broadcast2 n` — Similar to `broadcast n` except that provides a single shared input port; the message encapsulates the sender index and the data. Any data read on `inp` (and stripped from the emitter field) is atomically written on all `out#k`, for `k in 1..n-{`emitter`}`.

EXAMPLES

```
(* example: ring communication protocol with broadcasting medium: *)
needs "parrays";
fun ring1 s n = (* s is site's behavior; sites are indexed in 1..n *)
        (ARRAY (fn [n]=> s n) [n] /\ broadcast n) {/inp out};
```

```
fun ring2 s l = (* with state parameters passed to sites *)
        (PAR s l /\ broadcast (length l)) {/inp out};
```

SEE ALSO

Codec

CONTENTS

Various encoders/decoders, and generator of, for arbitrary bases (<256).

SIGNATURE

```
structure Codec =
   val mk_codec : string -> (int -> string) * (string -> int)
   val d’b : string -> int
   val b’d : int -> string
   val d’o : string -> int
   val o’d : int -> string
   val d’h : string -> int
   val h’d : int -> string
   val bools’byte : int -> bool array
   val bytes’int  : int -> int array
```

DESCRIPTION

`mk_codec alph` — String `alph` is a sequence of representing symbols of digits for numbers in base (`length alph`). Returns a pair of functions `(encoder, decoder)` that convert nonnegative decimals to/from the representation and base defined by `alph`.

`b’d, d’b` — decimal <-> binary converters.

`o’d, d’o` — decimal <-> octal converters.

`h’d, d’h` — decimal <-> hexadecimal converters.

`bytes’int n` — Returns the decomposition into bytes of integer $|n|$, stored into an array.

`bools’byte n` — Returns the decomposition into booleans of byte ($|n|$ `mod 256`) in base 2, stored into an array.

EXAMPLES

```
- b'd 256;
"100000000"
- d'h "61a80";
val it = 400000 : int
- o'd it;
val it = "1415200" : string

- open Array;
- bytes'int 46565633;
val it = [|2, 198, 137, 1|] : int array
- bools'byte (Array.sub (it,1));
val it = [|true, true, false, false, false, true, true, false|] : bool array
```

SEE ALSO

Dataflow

CONTENTS

Some behaviors and combinators for (extended) “dataflow” parallel programming.

SIGNATURE

```
structure Dataflow =
   val collect : {inp: 'a}_b
   val naturals : {out: int}_a
   val generatef : ('a -> 'a) -> 'a -> {out: 'a}_b
   val mapf : ('a -> 'b) -> {inp: 'a, out: 'b}_c
   val filterf : ('a -> bool) -> {inp: 'a, out: 'a}_b
   val findf : ('a -> bool) -> {inp: 'a, out: 'a}_b
   val splitf : {inp: 'a * 'b, out1: 'a, out2: 'b}_c
   val combinef : {inp1: 'a, inp2: 'b, out: 'a * 'b}_c
   val merge : {inp1: 'a, inp2: 'a, out: 'a}_b
   val distr : {inp: 'a, out1: 'a, out2: 'a}_b
   val buffer : 'a list -> {inp: 'a, out: 'a}_b
   val flow_of_list : 'a list -> 'a -> {out: 'a}_b
   val list_of_flow : ('a -> bool) -> {inp: 'a, out: 'a list}_b
   val irandom : int -> {out: int}_a
   val brandom : {out: bool}_a
   val prandom : int -> {out: bool}_a
```

DESCRIPTION

`collect`	A behavior that consumes forever on port `inp`.
`naturals`	Offers integers in increasing order on port `out`.
`generatef f a`	Offers successively `a, f a, f (f a)`, etc, on port `out`.
`mapf f`	Acts as a filter, takes input x on `inp`, offers `f x` on `out`, repeatedly.
`filterf p`	Takes input on `inp`, offers it on `out` if satisfies `p`, repeatedly.
`findf p`	Takes input on inp, until one is read that satisfies `p`. The value read is then offered on `out`.
`splitf`	Inputs pairs of values on `inp`, asynchronously offers their `fst` on `out1` and their `snd` on `out2`, repeatedly.
`combinef`	Asynchronously inputs a value `x` on `inp1` and a value `y` on `out2`, offers then `(x,y)` on `out`, repeatedly.
`merge`	Inputs either on `inp1` or `inp2`, and offers the value read on `out`, repeatedly.
`distr`	Inputs on `inp`, offers the value read on either `out1` or `out2`, repeatedly.
`buffer b`	Asynchronously inputs on `inp` and offers the value read on `out`, repeatedly (no size bound for the bufferized queue).
`flow_of_list l e`	Offers elements of list `l` on `out`, then value `e` forever when list `l` is exhausted.

`list_of_flow p`	Accumulates in a list the elements input on port `inp`, while they satisfying predicate p. Offers list on port `out` then, and stops.
`irandom n`	Offers a random integer in range `0..(n-1)` on `out`, repeatedly.
`brandom`	Offers a random boolean on `out`, repeatedly.
`prandom k`	k is an integer in range `1..100`. Offers a random boolean on `out`, repeatedly, `true` with probability `k/100`.

EXAMPLES

A pipelined prime number generator:

```
- agent primes =
= let fun filterp p =             (* filters non multiples of p *)
=        filterf  (fn x => x mod p <> 0)
=     agent sieve =               (* offers primes *)
=        inp? x => out! x => (filterp x ^^ sieve)
= in (naturals ^^ inp? x=> out! x=> inp? x=> out! x=> sieve) {/inp}
= end;
val primes = - : {out: int}_a
```

SEE ALSO

examples/dataflow/*

Extras

CONTENTS

General purpose additional structures.

SIGNATURE

```
structure Extras =
   structure Rat = ...
   structure Mprat = ...
   structure Alist = ...
   structure Slist = ...
   structure Tlist = ...
   structure Lizt = ...
   structure Stack = ...
   structure Queue = ...
   structure Stree = ...
   structure Avltree = ...
   structure Time = ...
   structure Print = ...
   structure Term = ...
   structure Stdterm = ...
   structure Bench = ...
   structure Codec = ...
   structure Sort = ...
   (* next six omitted if process option is -n *)
   structure Timing = ...
   structure Broadcast = ...
   structure Dataflow = ...
```

```
structure Parrays = ...
structure Pio = ...
```

DESCRIPTION

A structure that packs together all structures in this section (except itself). It is loaded and installed by reading in script `lextras` (see Section 3).

EXAMPLES

SEE ALSO

script `lextras`.

Lizt

CONTENTS

Lazy lists.

SIGNATURE

```
structure Lizt =
   datatype 'a lizt
   con nils : 'a lizt
   con ::: : 'a * 'a lizt lift -> 'a lizt
   exception Hs : exn
   val hs : 'a lizt -> 'a
   exception Ts : exn
   val ts : 'a lizt -> 'a lizt
   val nulls : 'a lizt -> bool
   val generates : ('a -> 'a) -> 'a -> 'a lizt
   val maps : ('a -> 'b) -> 'a lizt -> 'b lizt
   val filters : ('a -> bool) -> 'a lizt -> 'a lizt
   exception Nths : exn
   val nths : 'a lizt -> int -> 'a
   val splits : ('a * 'b) lizt -> 'a lizt * 'b lizt
   exception Combines : exn
   val combines : 'a lizt * 'b lizt -> ('a * 'b) lizt
```

DESCRIPTION

`'a lizt`	A type of lazy lists. Type `lizt` does not admit equality.
`nils`	The empty lazy list.
`:::`	Consing constructor for lizts.
`hs` l	Returns the head of `l`; raises `Hd` if `l` empty.
`ts` l	Returns the tail of `l`; raises Ts if `l` empty.
`nulls` l	Returns true if `l` empty.
`generates` f e	Returns the lazy list `[e, f e, f (f e), ...]`
`maps` f l	Applies `f` to all elements of lizt `l`.
`filters p l`	Returns `l` stripped from its elements not satisfying predicate `p`.

`nths l n`	Returns the n^{th} component of `l`. Raises `Nth` if `l` has less than n components.
`splits` l	Components of lizt `l` are pairs, returns the pair `(l1,l2)` in which `l1` holds the `fst` of the elements of `l`, and `l2` holds the `snd` of these.
`combines` (l1,l2)	Inverse of `splits`. Builds a lizt of pairs from a pair of lizts.

EXAMPLES

Some examples to play with:

```
- val naturals = generates (fn x => x + 1) 0;
val naturals = 0 ::: up - : int lizt

- val squares = maps (fn x => x*x) naturals;
val squares = 0 ::: up - : int lizt

- val fibs = (* first form; using an intermediate recursive function *)
=  let fun from (x,y) = let val z = x + y in z ::: up (from (y,z)) end
=  in 1 ::: up (1 ::: up (from (1,1))) end;
val fibs = 1 ::: up - : int lizt

- val fibs2 = (* second form; with mutually recursive lizts *)
=  let fun adds z1 z2 = maps (op + : int * int -> int) (combines (z1,z2))
=      lazy    z1 = up (1 ::: z2)
=      and     z2 = up (1 ::: z3)
=      and     z3 = up (adds (down z1) (down z2))
=  in down z1
=  end;
val fibs2 = 1 ::: up - : int lizt

- nths fib2 40;
val it = 102334155 : int
```

SEE ALSO

R. Cartwright, J. Donahue, “The Semantics of Lazy (and Industrious) Evaluation”, 1982 ACM Symposium on Lisp and Functional Programming. ACM 1982.

Mprat

CONTENTS

Multiple precision rational numbers.

SIGNATURE

```
structure Mprat =
   structure Rep = struct
      datatype mprat   [mprat]
      con // : mp * mp -> mprat
   datatype mprat   [mprat]
   con // : mp * mp -> mprat
```

```
val over : mp * mp -> mprat
val ~ : mprat -> mprat
exception Inv : exn
val inv : mprat -> mprat
val + : mprat * mprat -> mprat
val - : mprat * mprat -> mprat
val * : mprat * mprat -> mprat
val / : mprat * mprat -> mprat
val < : mprat * mprat -> bool
val <= : mprat * mprat -> bool
val > : mprat * mprat -> bool
val >= : mprat * mprat -> bool
val eq : mprat * mprat -> bool

val string_of_mprat : mprat -> string
exception Mprat_of_string : exn
val mprat_of_string : string -> mprat
val makestring : mprat -> string
val print : mprat -> mprat
```

DESCRIPTION

Multiple precision rational numbers and rational arithmetics. Same interface as `Rat`, except for the missing `real<->rat` conversions.

`mprat`	Type of arbitrary precision rational numbers. Type `mprat` admits equality.
`n over d, n//d`	Mprats are built by function `over` or constructor `//`. `//` does not normalize them, `over` do.
`Rep`	Representation structure.Provided to allow users to easily switch between `rat/mprat //` (since constructors cannot be overloaded). Opening `Mprat.Rep` (resp. `Rat.Rep`) makes visible the `mprat` (resp. `rat`) version of `//`.
`~, +, -, *, /`	Arithmetic operations on rationals. `/` raises `Inv` if result is invalid.
`inv r`	Inverse of r, i.e. `inv (n//d) = d//n`. Raises `Inv` if invalid.
`>, <, >=, <=`	Comparison functions.
`eq`	Equality. `eq` and = coincide on rationals in normal forms (as produced by `over` or by arithmetic operators).
`string_of_rat r`	Returns a string representing rational `r`.
`rat_of_string s`	Returns the `rat` represented by string `s`. Raises `Rat_of_string` if `s` does not represent a rational.
`print r`	Identity function. Prints representation of `r`.
`makestring r`	String representation of `r`.

EXAMPLES

```
- open Mprat.Rep;
datatype mprat   [mprat]
con // : mp * mp -> mprat
```

```
- 012344 over 02432426;
val it = 06172 // 01216213 : mprat

- 0234//04664 * 0455366363//04132324;
val it = 053277864471 // 09636579568 : mprat

- inv it;
val it = 09636579568 // 053277864471 : mprat
```

SEE ALSO

`Prelude.Mp`, `Extras.Rat`

Parrays

CONTENTS

Agent combinators for building arrays of communicating processes.

SIGNATURE

```
structure Parrays =
   val ARRAY : (int list -> _a) -> int list -> _a
   val SqMatrix : (int list -> _a) -> int -> _a
   val Matrix : (int list -> _a) -> int * int -> _a
   val Hypercube : (int list -> _a) -> int -> _a
```

DESCRIPTION

`ARRAY c D` — Agent `ARRAY` organizes a set of processes (the cells of the array) into an hyperarray. The behavior c of individual cells is determined by their index in the structure. D holds the sizes of the array in its different planes.

`SqMatrix c n, Matrix c (n,m), Hypercube c n`

Specializations of ARRAY for square matrices, matrices, and hypercubes.

EXAMPLES

SEE ALSO

Pio

CONTENTS

Behavioral interfaces to standard window io, files, windows.

SIGNATURE

```
structure Pio =
   val Stdin    : {stdin: string}_a
   val Stdout   : {stdout: string}_a
   val Stdio    : {stdin: string, stdout: string}_a
   val infile   : string -> {off, on, out: string}_a
   val outfile  : string -> {inp: string, off, on}_a
```

```
   val terminal : string -> {ctr: string, inp: string, off, on, out: string}_a
```

DESCRIPTION

`Stdin, Stdout, Stdio`	Agent `Stdout` provides a port `stdout`; any string sent to that port will be printed on `std_output()` as a side-effect. `Stdin` provides a symmetric facility on port `stdin`. Note that reading on `stdin` may be ambiguous if done from an agent running in background, as some other processes may, in parallel, read on `std_input()` too (e.g. the top-level process).
`infile f, outfile f`	`Agent infile` and `outfile` provide process abstractions of host operating system files. They provide ports for opening, performing io and closing files.
`terminal l`	Behavioral interface to interacting windows. Provide ports

EXAMPLES

```
open Pio;
start terminal "xterm";
start on! => inp!"hello" => out?l => off! sleep 1.0 => stop;
```

SEE ALSO

Print

CONTENTS

Pretty-printing package.

SIGNATURE

```
structure Print =
   exception Print : exn

   val pretty_printing : bool -> unit
   val print_depth : int -> unit
   val print_width : int -> unit

   val print_init : unit -> unit
   val print_begin : bool * int -> unit
   val print_token : string -> unit
   val print_break : int * int -> unit
   val print_end : unit -> unit
   val print_newline : unit -> unit
   val print_line : string -> unit

   val print_unit : unit -> unit
   val print_bool : bool -> unit
   val print_int : int -> unit
   val print_mp : mp -> unit
   val print_real : real -> unit
   val print_string : string -> unit
```

```
val print_list : bool * bool -> string * string * string -> ('a -> 'b) -> 'a list -> unit
```

DESCRIPTION

Structure `Print` provides pretty-printing facilities based upon “Prettyprinting, by D.C. Oppen, ACM Toplas Vol. 2, No 4, October 1980”.

`pretty_printing b`	Turns pretty-printing on/off.
`print_depth d`	Sets max block-depth of printing to d.
`print_width w`	Sets page width to w.
`print_init ()`	Initializes the pretty printer.
`print_begin (b,i)`	Starts printing a block with breaks aligned iff b true and indentation i.
`print_token s`	Pretty-prints the token represented by string s.
`print_break (i,off)`	Prints a break with indentation i and offset off
`print_end ()`	Ends printing of a block. Calls of functions `print_begin` and `print_end` must be correctly nested; no `print_line`, nor any pretty-printing control function should appear within a block.
`print_newline ()`	Prints a return and re-initializes printer
`print_line s`	Prints a string; then a return, and re-initializes printer

`print_unit (), print_bool b, print_int n, print_mp m, print_real r, print_string s`
Printers for special constants

`print_list (b,c) (x,y,z) p l`

Prints the list l with breaks aligned iff b is true. If c is true, then all items are printed at the same depth, otherwise depth is increasing with rank in list l. x is the “list-begin” character, y separates components of the list, z concludes the list, and p is a printer for the components of the list.

EXAMPLES

SEE ALSO

"Prettyprinting", by D.C. Oppen, ACM Toplas Vol. 2, No 4, October 1980.

Queue

CONTENTS

A queue abstract type.

SIGNATURE

```
structure Queue =

   abstype 'a queue
   exception Deq : exn
   val empty   : 'a queue
   val isempty : 'a queue -> bool
   val enq     : 'a -> 'a queue -> 'a queue
   val deq     : 'a queue -> 'a * 'a queue
   val head    : 'a queue -> 'a
```

```
abstype 'a iqueue
val newqueue     : unit -> '1a iqueue
val clearqueue   : 'a iqueue -> unit
val isemptyqueue : 'a iqueue -> bool
val enqueue      : 'a -> 'a iqueue -> unit
val dequeue      : 'a iqueue -> 'a
val headqueue    : 'a iqueue -> 'a

type 'a oqueue = (unit -> unit) *
                 (unit -> bool) *
                 ('a -> unit) *
                 (unit -> 'a) *
                 (unit -> 'a)
val newoqueue : unit -> '1a oqueue
```

DESCRIPTION

Functional interface:

`'a queue`	Abstract type of (functional) queues. Type `queue` does not admit equality.
`empty, isempty q`	Returns an empty queue, returns true if queue `q` is empty.
`enq e q`	Returns queue q with element e appended at the end.
`deq q`	Removes the first of `q`, returns it paired with the remaining queue. Raises `Deq` if `q` is empty.
`head q`	Returns the first of `q`. Raises `Deq` if `q` is empty.

Imperative interface:

`'a iqueue`	Abstract type of queues with state. `iqueues` are references to queues, this allows to omit the resulting queues in return types of primitives.
`newqueue ()`	Returns a new `iqueue`.
`clearqueue iq`	Clears `iq`.
`isemptyqueue iq`	True if `iq` empty.
`enqueue e iq`	Appends element e to `iq` as a side-effect.
`dequeue iq`	Returns head of `iq`, dequeues `iq` as a side-effect. Raises `Deq` if `iq` empty.
`headqueue iq`	Returns head of iq, does not dequeue it. Raises `Deq` if `iq` empty.

Private-primitives interface:

`'a oqueue`	Abstract type of queues with private primitives.
`newoqueue ()`	Creates an `oqueue` and returns a tuple of functions (clear, isempty, enqueue, dequeue, head) to manipulate the queue. These functions encapsulate the queue, they act like those in the `iqueue` interface, except that they implicitly operate on the `oqueue` created.

EXAMPLES

SEE ALSO

Rat

CONTENTS

Signed, fixed precision, rational numbers (with Daniel Loeb, July 1990).

SIGNATURE

```
structure Rat =
   type rat
   structure Rep = struct
      con // : int * int -> rat
      val over : int * int -> rat
   con // : int * int -> rat
   val over : int * int -> rat
   val ~ : rat -> rat
   exception Inv : exn
   val inv : rat -> rat
   val + : rat * rat -> rat
   val - : rat * rat -> rat
   val * : rat * rat -> rat
   val / : rat -> rat -> rat
   val eq : rat * rat -> bool
   val > : rat * rat -> bool
   val < : rat * rat -> bool
   val >= : rat * rat -> bool
   val <= : rat * rat -> bool

   val real'rat : rat -> real
   val rat'real : real -> rat
   val rat2'real : int -> real -> rat

   val string_of_rat : rat -> string
   exception Rat_of_string : exn
   val rat_of_string : string -> rat
   val print : rat -> rat
   val makestring : rat -> string
```

DESCRIPTION

Rational numbers and rational arithmetics.

`rat`	Type of rational numbers. Type `rat` admits equality.
`n over d, n//d`	Rational number are built by function `over` or constructor `//`. `//` does not normalize them; `over` do. `Rat`'s rationals are finite precision, numerator and denominator must be in range `Int.minint..Int.maxint`.
`Rep`	Representation structure.Provided to allow users to easily switch between `rat/mprat //` (since constructors cannot be overloaded). Opening `Mprat.Rep` (resp. `Rat.Rep`) makes visible the `mprat` (resp. `rat`) version of `//`.
`~, +, -, *, /`	Arithmetic operations on rationals. `/` raises `Inv` if result is invalid.
`inv r`	Inverse of r, i.e. `inv (n//d) = d//n`. Raises `Inv` if invalid.

`>, <, >=, <=`	Comparison functions.
`eq`	Equality. `eq` and = coincide on rationals in normal form (as produced by `over` or by arithmetic operators).
`real’rat ra`	`rat` to `real` conversion (approximate).
`rat’real re`	`real` to `rat` conversion (approximate). Automatic precision adjustment to avoid numerator or denominator `Overflow.`
`rat2’real p re`	`real` to `rat` conversion (approximate). Explicit precision.
`string_of_rat r`	Returns a string representing rational `r`.
`rat_of_string s`	Returns the `rat` represented by string `s`. Raises `Rat_of_string` if `s` does not represent a rational.
`print r`	Identity function. Prints representation of `r`.
`makestring r`	String representation of `r.`

EXAMPLES

```
- 12//234 / 4543//34;
val it = 68 // 177177 : rat

- real’rat it;
val it = 0.000383796993966486 : real

- map (fn x => rat2'real x pi) [1,2,3,4,5];
val it = [3 // 1, 22 // 7, 333 // 106, 355 // 113, 103993 // 33102] : rat list

- rat2'real 16 pi;
uncaught exception  --  Overflow

- rat’real pi;
val it = 817696623 // 260280919 : rat
```

SEE ALSO

`Prelude.Mp`, `Extras.Mprat`

Slist

CONTENTS

Sets as lists.

SIGNATURE

```
structure Slist =
   val distinct : ''a list -> bool
   val make_set : ''a list -> ''a list
   val intersect : ''a list -> ''a list -> ''a list
   val subtract : ''a list -> ''a list -> ''a list
   val symdiff : ''a list -> ''a list -> ''a list
   val union : ''a list -> ''a list -> ''a list
```

```
    exception Repl : exn
    val repl : 'a -> int -> 'a list
    val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
    val range : int -> int -> int list
    val count : ('a -> bool) -> 'a list -> int
```

DESCRIPTION

`distinct l`	True if all elements in list l are different (by Bool.<>).
`make_set l`	Makes a list into a set, stripping out duplicate elements
`intersect l1 l2`	Set intersection of lists l1 and l2.
`union l1 l2`	Set union of lists l1 and l2.
`subtract l1 l2`	Set difference of lists l1 and l2.
`symdiff l1 l2`	Set symmetric difference of lists l1 and l2.
`repl x n`	Returns the list [x; x; ...; x] of length n
`range from to`	Returns the int list [from; from+1; ...; to]
`partition p l`	Returns pair (l1,l2) in which l1 (resp. l2) holds the items of l satisfying (resp. not satisfying) predicate p
`count p l`	Number of elements of l that satisfy p (eqv. length (filter p l)).

EXAMPLES

SEE ALSO

Sort

CONTENTS

Integer lists sorting functions.

SIGNATURE

```
structure Sort =
   val isort : 'a list -> ('a * 'a -> bool) -> 'a list
   val qsort : 'a list -> ('a * 'a -> bool) -> 'a list
```

DESCRIPTION

`qsort l sup`	Sorts list l in increasing order according to function sup. Uses quicksort algorithm.
`isort l sup`	Sorts list l in increasing order according to function sup. Uses insertion sort algorithm.

EXAMPLES

SEE ALSO

Stack

CONTENTS

Stacks, various forms.

SIGNATURE

```
structure Stack =

   abstype 'a stack
   exception Pop : exn
   val new   : 'a stack
   val isnew : 'a stack -> bool
   val push  : 'a -> 'a stack -> 'a stack
   val pop   : 'a stack -> 'a * 'a stack
   val top   : 'a stack -> 'a

   abstype 'a istack
   val newstack   : unit -> '1a istack
   val isnewstack : 'a istack -> bool
   val clearstack : 'a istack -> unit
   val pushstack  : 'a -> 'a istack -> unit
   val popstack   : 'a istack -> unit
   val topstack   : 'a istack -> 'a

   type 'a ostack = (unit -> unit) *
                    (unit -> bool) *
                    ('a -> unit) *
                    (unit -> 'a) *
                    (unit -> 'a)
   val newostack : unit -> '1a ostack
```

DESCRIPTION

Functional interface:

`'a stack`	Abstract type of (functional) stacks. Type `stack` does not admit equality.
`new, isnew s`	Returns an empty stack, returns true if stack `s` is new.
`push e s`	Returns stack q with element e pushed on it.
`pop s`	Removes the top element of s, returns it paired with the remaining stack. Raises `Pop` if `s` is new.
`top s`	Returns the top of `s`. Raises `Pop` if `s` is new.

Imperative interface:

`'a istack`	Abstract type of stacks with state. `istacks` are references to stacks, this allows to omit the resulting stacks in return types of primitives.
`newstack ()`	Returns a new `istack`.
`clearstack is`	Clears `is`.
`isnewstack is`	True if `is` new.
`pushstack e is`	Pushes element e onto `is` as a side-effect.

`popstack is`	Returns top of `is`, pops `is` as a side-effect. Raises `Pop` if `is` new.
`topstack is`	Returns top of is, does not pop it. Raises `Pop` if `is` new.
Private-primitives interface:	
`'a ostack`	Abstract type of stacks with private primitives.
`newostack ()`	Creates an `ostack` and returns a tuple of functions (clear,is-new,push,pop,top) to manipulate the stack. These functions encapsulate the stack, they act like those in the `istack` interface, except that they implicitly operate on the `ostack` created.

EXAMPLES

SEE ALSO

Stdterm

CONTENTS

Terminal control utilities [preliminary version].

SIGNATURE

```
structure Termio =
   val clear : unit -> unit
   val beep : unit -> unit
   val home : unit -> unit
   val move : int * int -> unit
```

DESCRIPTION

Specialization for std_window() of some functions in structure Term:

`clear ()`	Clears standard window of user interface.
`beep ()`	Beeps standard window of user interface.
`home ()`	Moves cursor to home position (without clear).
`move (x,y)`	Moves cursor to position (x,y).

EXAMPLES

SEE ALSO

Stree

CONTENTS

Binary search trees with string keys.

SIGNATURE

```
structure Stree =
  abstype 'a stree
  val nullstree : 'a stree
```

```
  exception Lookupstree : exn
  val lookupstree : 'a stree -> string -> 'a
  val updatestree : 'a stree * (string * 'a) -> 'a stree
  val streedepth : 'a stree -> int
  val balancestree : 'a stree -> 'a stree
  val list_of_stree : 'a stree -> (string * 'a) list
```

DESCRIPTION

`'a stree`	Abstract type of binary search trees with string keys. Interface is identical to that of structure Avltree. Type `stree` does not admit equality.
`nullstree`	Returns an empty stree.
`lookupstree` t k	Returns the item associated with key k in tree t. Raises `Lookupstree` if none is found.
`updatestree t (k,v)`	If some item is stored under key k in tree t, then replaces the item by v. Otherwise the entry (k,v) is added to the tree. Strees are built from `nullstree` and `updatestree`.
`streedepth t`	Returns the depth of the deepest item in tree t.
`balancestree t`	Balances tree t.
`list_of_stree t`	Builds a list from all pairs (key,item) stored in tree t, in increasing lexicographic order of keys.

EXAMPLES

SEE ALSO

Term

CONTENTS

Terminal capabilities [preliminary version].

SIGNATURE

```
structure Term =
  exception Terminfo : exn -> string
  val clear_win : window -> unit
  val beep_win : window -> unit
  val delchar_win : window -> unit
  val move_win : window -> int * int -> unit
  val home_win : window -> unit
  val rvideo_win : window -> unit
```

DESCRIPTION

`clear_win w`	Clears window w.
`beep_win w`	Beeps window w (bell or visual, according to window setting.
`delchar_win w`	Deletes character at current cursor position in window w.
`move_win w (x,y)`	Moves cursor in window w to position (x,y).
`home_win w`	Moves cursor in window w to home position (no clear).

`rvideo_win w`	Sets window w in reverse video mode.

EXAMPLES

```
- clear_win (std_window ());
val it = () : unit

- val w = open_win ();
val w = - : window

- rvideo_win w;
val it = () : unit

- output_win (w,"hello\n");
val it = () : unit

- clear_win w;
val it = () : unit
```

SEE ALSO

Time

CONTENTS

Timing functions.

SIGNATURE

```
structure Time =
   val start_timer : unit -> int * int
   val check_timer : int * 'a -> int
   val check_timer_gc : 'a * int -> int
   val time : ('a -> 'b) -> 'a -> 'b
```

DESCRIPTION

`start_timer ()`	Returns times spent in execution and garbage collection (in ms) in the current lcs session.
`check_timer (r,_)`	Returns r - time spent in execution (in ms) in session. If the argument is the result of `start_timer`, then this is the time spent in gc since last call of `start_timer`.
`check_timer_gc (_,g)`	Same as above, except for garbage collection time.
`time f x`	Evaluates (f x), and prints as side-effect the time spent in execution and gc for its evaluation.

EXAMPLES

SEE ALSO

`Prelude.System`: built-in `timer` and `timers` functions.

Timing

CONTENTS

Watchdogs and other time-related behaviors.

SIGNATURE

```
structure Timing =
   val clock : real -> {tick, tock}_a
   val watch : real -> _a -> _a -> _a
   val timeout : real -> {out: 'a, tmp.''b: 'c}_d -> {out: 'a}_d -> {out: 'a}_d
   val ping : real -> {hello: 'a}_b -> {hello: 'a}_b -> {hello: 'a}_b
```

DESCRIPTION

`clock n`	Offers `()` on port `tick` every $\lvert n \rvert$ seconds; stopped by sending `()` on port `tock`.
`watch d a b`	Proceeds with `b` if `a` does not move within delay `d`.
`timeout d a b`	Proceeds with `b` if `a` does not offer a value on `out` within delay `d`.
`ping d a b`	Proceeds with `b` if `a` does not send periodic message on port `hello`.

EXAMPLES

SEE ALSO

Tlist

CONTENTS

Tables as lists.

SIGNATURE

```
structure Tlist =
   exception Range : exn
   val chop : int -> 'a list -> 'a list * 'a list
   val nth : 'a list -> int -> 'a
   val dropnth : int -> 'a list -> 'a list
   val replacenth : int -> 'a -> 'a list -> 'a list
   val pack : int -> 'a list -> 'a list list
   val transpose : 'a list list -> 'a list list
```

DESCRIPTION

Tables as lists (vectors, matrices, etc).

`chop n l`	chops l into two lists (l1,l2), with length l1 = n. Raises `Range` if not possible.
`nth l n`	Returns n^{th} element of `l`. Raises `Range` if n not in range `1..(length l)`.
`replacenth n e l`	Replaces the nth element of l by e. Raises `Range` if n out of range.
`dropnth n l`	Removes the nth element of l. Raises `Range` if n out of range.
`pack n l`	Packs list l into a list of rows of n elements. Raises `Range` if not possible.

`transpose l`	Transposes list of lists l (list of rows => list of columns).

EXAMPLES

```
- pack 2 [1,2,3,4,5,6];
val it = [[1, 2], [3, 4], [5, 6]] : int list list

- transpose it;
val it = [[1, 3, 5], [2, 4, 6]] : int list list

- dropnth 1 it;
val it = [[2, 4, 6]] : int list list

- flat it;
val it = [2, 4, 6] : int list

- replacenth 1 10 it;
val it = [10, 4, 6] : int list
```

SEE ALSO

2.4 Network: distributed programming

This section describes structure `Networks` and all of its sub-structures. Structure `Network` is not initially loaded; it can be loaded and installed by reading in script `lnetwork` (see Section 3). Its definition is found in the sequel, followed by the definitions of all of its sub-structures.

Dest

CONTENTS

Type destination, plus termination event.

SIGNATURE

```
structure Dest =
   datatype dest   [dest]
   con All : dest
   con To : int -> dest
   con Not : int -> dest
   con Som : int list -> dest
   event TERM : unit evt
```

DESCRIPTION

`dest`	Type of message destinations in network abstractions Star, Ptop, and Ring. Sites are identified by integers, `dest` values encode some sets of integers. Possible destinations of a message are, for some `n` and `l`:
`All`	All-sites destination.
`To n`	Site n only.
`Not n`	All sites except n.
`Som l`	All sites in list l.
`TERM`	Event used to signal termination of network clients.

EXAMPLES

SEE ALSO

`Network.Star, Network.Ptop, Network.Ring.`

Go

CONTENTS

Support for remote processing.

SIGNATURE

```
structure Go =
   exception Time : exn
   val go : _ service name -> real -> ('a -> '4b) -> 'a -> '4b
   val go_server : _ service name -> _
```

DESCRIPTION

Defines server and client interfaces for remote processing facilities. Server runs as a daemon. Clients send to server pairs (f,x), server evaluates (f x) and returns the result to client.

`go n d f x`	Evaluates (f x) on the remote application offering the evaluation service identified by name `n`. Raises `Time` if evaluation could not be completed in `d` seconds.
`go_server` n	Behavior of the remote evaluation server. Client's code is executed in the highest level protected mode (see `System.Protect` and flag "-p" of the lcs command for details on protection levels).

EXAMPLES

Creating a remote evaluation daemon and application:

1. Create a `submit` function and the associated `server` behavior, save heap image, then launch evaluation daemon by:

```
here% lcs
- use "go";
- open Go;
- val (submit,server) =
=     let val n = Register.Name.name ()
=     in (go_client n, go_server n)
=     end;
- save "go";
- reboot server;
^Z
here% bg
```

2. Invoke go daemon from the heap image saved, as follows (e.g.):

```
there% go
- submit 0.5 (op div) (6,2);
val it = 3 : int
- submit 0.5 (op div) (6,0);
uncaught exception  --  Div
- submit 1.0 open_in "foo";
uncaught exception  --  Host ("security violation" : string)
- needs "bench";
- submit 10.0 Bench.fib 30;
val it = 1346269 : int
- submit 1.0 Bench.fib 30;
uncaught exception  --  Time
- ^D
there%
```

SEE ALSO

Move

CONTENTS

Support for (some) mobile applications.

SIGNATURE

```
structure Move =
   type move_msg
   val move : move_msg name -> (window -> _) -> _
   event MOVED : unit evt
```

DESCRIPTION

Structure Move allow to define applications that migrate to the site of the caller when invoked. The server opens and registers a connection service, starts an application, exports it when contacted, then terminates; the client contacts the server, imports a behavior, then becomes server.

`move n f`	The top level of "moving" applications. n is a name identifying the connection service, f is the behavior the application implements (taking a window as parameter).
`move_msg`	The names to be used as parameter of "moving" applications must have type `move_msg name`.
`MOVED`	Signals termination of move, emitted once transfer of behavior is complete.

Note on passing of io constants, exceptions and events between lcs applications:

- IO streams and windows have meanings local to an lcs machine (i.e. to a unix process), in particular, constant `std_win`, and the derived `std_in`, `std_out`, `std_ctrl` do differ between sites. So, moving applications that require some window to operate must be parameterized by an operating window (rather than capture it in the code); and the operating window must be passed to the application once its behavior has been imported, as done below.

- The same applies for non-primitive exceptions and events (those declared with **`exception`** or **`event`**, by the user of in some library file): nonprimitive exceptions and events have site-local meanings. So, for instance, don't expect to trap an exception raised from some imported code with a handler specified for a local exception (unless that exception is a primitive one).

- On another hand, port labels have site-independent meanings, agents have their interaction capabilities unchanged when exported.

EXAMPLES

See `medor` demo.

SEE ALSO

Mutex

CONTENTS

Network abstraction layer providing distributed mutual exclusion.

SIGNATURE

```
structure Mutex =
   datatype over_mode
   WAIT : real -> over_mode
   EXIT : real -> over_mode
   type 'a mutex_msg
   val mutex :
```

```
      over_mode ->
      {ago: int,                         (* application go indication   *)
       aind: int * 'a,                   (* application data indication *)
       aquit,                            (* application quit request    *)
       areq: dest * 'a,                  (* application data request    *)
       ngo: int,                         (* network go indication       *)
       nind: int * 'a mutex_msg,         (* network data indication     *)
       nquit,                            (* network quit indication     *)
       nreq: dest * 'a mutex_msg,        (* network data request        *)
       granted,                          (* token reply                 *)
       over,                             (* token release               *)
       token}_b                          (* token request               *)
   val push_mutex :
      over_mode ->
      {go: int, ind: int * 'a mutex_msg, quit, req: dest * 'a mutex_msg,
       granted, over, token}_b ->
      {go: int, ind: int * 'a, quit, req: dest * 'a,
       granted, over, token,
       ngo#''c: 'd, nind#''e: 'f, nquit#''g: 'h, nreq#''i: 'j}_b
```

DESCRIPTION

Operation: Distributed mutual exclusion is handled by a token circulating among mutex entities. A user entity (just above mutex level) requests the token by sending `()` on port `token`. The token is available on port `granted`, and it is released by sending `()` on port `over`. In parallel, the network traffic on ports `ngo` and `nreq` is echoed to the local user on ports `ago`, `areq`, and the traffic from the user on ports `aind`, `aquit` is passed to the network on ports `nind` and `nquit` (whether the token has been acquired or not). The mutex layer transparently multiplexes/demultiplexes the messages pertaining to the mutex protocol and those pertaining to communication between the entities of layers above mutex.

`'a mutex_msg`	Type of messages transferred between mutex entities.
`mutex` b ono	Behavior `mutex` is a layer to be pushed on top of a network abstraction (as defined by the `Star`, `Ptop`, and `Ring` structures), and that provide these with a distributed mutual exclusion service. The protocol implemented is symmetric (no privileged site), no provision is made to regenerate the mutual exclusion token when lost, and a client holding the token is not allowed to leave. Parameter ono encodes the termination strategy (see below).
`push_mutex` ono a	Combinator `push_mutex` ono pushes the mutual exclusion layer `mutex ono` on top of the network abstraction a (as defined by the `Star`, `Ptop`, or `Ring` structure). The combination extends the network interface with ports `granted`, `over`, and `token`, and refines the type of `ind` and `req` ports.
`over_mode`	Type encoding what to do when an entity wants to exit but owns the mutual exclusion token. One of the following:
`WAIT d`	`An` entity holding the token must pass it to another entity before leaving. Time `d` is the time period used to poll for incoming entities.
`EXIT d`	An entity holding the token is allowed to quit if it can't pass the token before `d` seconds elapsed. Note that this is equivalent to a token loss, if some new entity could manage to connect before the server was aware of termination of the token holder.

EXAMPLES

See `chat` demo for an example of use.

SEE ALSO

Network

CONTENTS

Structures for distributed programming.

SIGNATURE

```
structure Network =
   structure Socket = ...
   structure Remote = ...
   structure Sysregister = ...
   structure Register = ...
   structure Dest = ...
   structure Star = ...
   structure Ptop = ...
   structure Ring = ...
   structure Mutex = ...
   structure Simple = ...
   structure Go = ...
   structure Move = ...
```

DESCRIPTION

A structure that packs together all structures pertaining to distributed programming. It is loaded and installed by reading in script `lnetwork` (see Section 3).

EXAMPLES

SEE ALSO

Ptop

CONTENTS

Network abstraction; point to point architecture.

SIGNATURE

```
structure Ptop =
   type 'a group_msg
   val network : '1a group_msg name ->
                {go: int,                       (* go indication       *)
                 quit,                          (* quit request        *)
                 ind: int * '1a,                (* data indication     *)
                 req: dest * '1a}_b             (* data request        *)
```

DESCRIPTION

Network abstractions provide a behavioral abstraction for communication networks. Sites are numbered as they join the network, they may enter and leave the network at any time, and may exchange data with the other sites connected. Network abstractions `Star`, `Ptop`, and `Ring` provide the same interface but differ by their architecture. In the `Ptop` abstraction, the network server helps establish direct point-to-point links between each pair of clients, used to transfer site-to-site traffic.

`network n`	That behavior creates a network end (a client), with ports go, ind, req, and quit. The network is identified by name n. If the network is not registered yet, then it opens a server for the network, and then a client, both running in the same lcs machine, and registers the network. Otherwise, it simply opens a client and connects it to the existing network.
`'a group_msg`	The names used for creating networks must have type `ty group_msg name`, where `ty` is the type of data to be passed between sites; for type safety, type `ty` must be a monotype.

Network operation: Behavior `network n` first offers a client identifier on port `go` (an integer). Reading this identifier is mandatory, it allows the client's behavior to start. A client is terminated by sending it value `()` on port `quit`, the client acknowledges termination by emitting signal `TERM`. Data is sent to other clients connected to the network through port `req`, with a specified destination (see structure `Dest` for available destinations); data is received from other sites on port `ind`, with the identifier of the sender. Data transferred through the network must be monomorphic, this is essential for enforcing type safety.

EXAMPLES

SEE ALSO

Register

CONTENTS

Single-site service registration facilities.

SIGNATURE

```
structure Register =

   (* names *)
   structure Name =
      abstype 'a name
      val concat_name : 'a name -> string -> 'a name
      val equal_name  : 'a name * 'a name -> bool
      val name        : unit -> '1a name

   (* registration facilities *)
   structure Client =
      val get_service        : 'a service name -> 'a service
      val register_service   : 'a service name * 'a service -> unit
      val unregister_service : 'a service name -> unit
      val update_service     : 'a service name * 'a service -> unit

   (* registration server *)
   structure Server =
      val get_lcsd_service : unit -> rstore service
      val server : _
```

DESCRIPTION

Structure `Register` is a specialization of structure `Rstore` to create and use "service registration" daemons.

Names:

(see description of `Name` structure in `Rstore`)

Registration services:

Access and update of services database. In case of failure, these routines Raise exception Remote with a string parameter indicating the reason of failure.

`get_service n`	Returns service registered under name n.
`register_service (n,k)`	Register service k under name n.
`unregister_service n`	Removes all services registered under name n.
`update_service (n,k)`	Replaces by k the service registered under name n, if any.

Registration server:

`get_lcsd_service ()` Returns default lcs registration service.

`server` The behavior of the lcsd registration daemon, for documentation purposes (a specialization of `Rstore.server`).

The lcs command:

```
reboot do (dump ("lcsd","-q -b")) => Server.server;
```

Creates the standalone image of a registration server in the current directory and launches it.

The unix command:

```
lcsd {-r klebs.laas.fr 4500}
```

Re-launches the server just created on the host/port specified, or default host/port specified at installation of lcs command if arguments are omitted. `lcsd` fails if not launched from the host specified, or if an lcsd server is already running on that host.

EXAMPLES

SEE ALSO

Remote

CONTENTS

Inter-lcs polymorphic streams and secured remote connections services.

SIGNATURE

```
structure Remote =

   exception Remote : string -> exn

   (* connection services *)
   abstype 'a service
   val open_service : unit -> '1a service
   val close_service : 'a service -> unit
```

```
val open_pub_service : string * int * string -> '1a service
val get_pub_service : string * int * string -> '1a service
val write_service : 'a service -> string -> unit
val read_service : string -> '1a service
val de_service : 'a service -> Dev.key

(* streams *)
abstype 'a stream
val accept_connection : 'a service -> 'a stream
val open_connection : 'a service -> 'a stream
val close_connection : 'a stream -> unit
val ping_connection : 'a service -> bool
val timed_accept_connection : real -> 'a service -> {out: 'a stream}_b
val timed_open_connection : real -> 'a service -> {out: 'a stream}_b
val timed_ping_connection : real -> 'a service -> {out: bool}_b
val import : 'a stream -> 'a
val export : 'a stream * 'a -> unit
val de_stream : 'a stream -> Dev.dev
```

DESCRIPTION

Structure `Remote` provides primitives to establish connections between lcs applications. The interface is based upon TCP-IP sockets (connected mode). Compared to the raw connection primitives in structure `Socket`, those here provide some authentication, server and client connections exchange passwords. Structure `Remote` also provides polymorphic imput and output functions for streams (functions `import` and `export`, respectively).

WARNING: One must either use the function here, or those in structure `Socket`, but DO NOT mix both in the same application as this might break type security.

`Remote`	This exception is raised by many functions in this structure, to signal exceptional conditions.

Services:

`'a service`	The type of connection services. Do not admit equality. A service encapsulates an lcs device (encoding a TCP socket), a host name, a port number, and a password. Services must be created monomorphic.
`open_service ()`	Creates a connection service on local host, the port number and password are automatically allocated.
`close_service s`	Closes connection service s.
`open_pub_service (h,p,w)`	Creates a connection service with parameters made explicit: host h, port number p, and password w.
`get_pub_service (h,p,w)`	Returns the connection service identified by host h, port p, and password w. Does not create the service.
`write_service s f`	Saves service s in file f.
`read_service f`	UNSAFE. Retrieves service from file f, gives it any type (by type cast).
`de_service`	Breaks up a service into its constituents.

Streams:

`'a stream`	Type of bidirectional polymorphic inter-lcs streams. Do not admit equality. Streams are weakly polymorphic.

`accept_connection s`	Creates a stream-end from service s. Function `accept_connection` must be called from the process in which service s was created, conventionally called the server side.
`open_connection s`	Creates a stream-end from service s, client side.
`close_connection s`	Closes a stream-end.
`ping_connection s`	Returns true if service s is active, or false otherwise.
`timed_accept_connection d s`	Behavioral version of `accept_connection`. Offers a stream on port `out` and terminates once a connection is successfully accepted. Delay d is used to force termination of authentication in case the other end does not send its password within d seconds.
`timed_open_connection d s`	Behavioral version of `open_connection`. Offers a stream on port `out` and terminates once the connection is successfully created. Raises signal `EXN` with message `Remote "autentication failed"` if authentication is not completed within `d` seconds.
`timed_open_connection d s`	Behavioral version of `accept_connection`. Offers a boolean on port `out` and terminates once a connection is successfully tested. Raises signal `EXN` with message `Remote "ping failed"` if authentication is not completed within `d` seconds.
`import s`	Imports a value from stream `s`.
`export (s,v)`	Sends value `v` over stream `s`.
`de_stream s`	Breaks up a stream into its constituents.

NOTE: Most of the above functions should not be interrupted. Interruption of a function operating on a service or stream (by keyboard interrupt or lcs signals) may leave some streams locked, which makes them unavailable for further IO operations. This should be improved in latter releases of the library.

EXAMPLES

The following are two simple examples of connection establishment between two lcs applications. In both case, c stream connecting the applications is established, the factorial function is passed over it and is used in the receiving site. The first example uses "public" connection services, the second uses anonymous services together with the service registration facilities provided by structure `Register`. The left column is the server side, the right column is the client side.

1. *Connecting two lcs applications, the type-unsafe way*:

A "public" connection service is identified by a host name (here "`tippi.laas.fr`"), a port number (here `4501`), and a password (here "`hello`"). The server side creates the service by `open_pub_service`, the client side uses a handle to that service, returned by `get_pub_service`.

```
unix1% lcs                          unix%2 lcs
val it = () : unit                  val it = () : unit

- prompt ("\n1> ","1= ");           - prompt ("\n2> ","2= ");
val it = () : unit                  val it = () : unit

1> needs "lnetwork";                2> needs "lnetwork";
```

```
1> val k = open_pub_service
1= ("tippi.laas.fr",4501,"hello")
1= : (int -> int) service;
val k = - : (int -> int) service
                                    2> val k = get_pub_service
                                    2= ("tippi.laas.fr",4501,"hello")
                                    2= : (int -> int) service;
                                    val k = - : (int -> int) service
1> val s = accept_connection k;
                                    2> val s = open_connection k;
                                    val s = - : (int -> int) stream
val s = - : (int -> int) stream

1> needs "bench";
1> export (s,Bench.fact);
val it = () : unit
                                    2> import s 5;
                                    val it = 120 : int

1> ^D                               2> ^D
unix1%                              unix%2
```

2. *Connecting two lcs applications, the type-safe way*:

Public services need not be registered in some service database since a handle to an existing service can be built from the service parameters (by `get_pub_service`) from any application. Function `get_pub_service` is not type-safe though; nothing relate the types of `open_pub_service` and `get_pub_service`, when their parameters (and thus the services referred to) are the same. A type-safe alternative (mandatory in the case of anonymous services created by `open_service`) is to use the type safe registration facilities provided by structure `Register` instead.

The technique consists of pre-declaring a name with function `Register.Name.name`, under which the connection service to be later created will be registered, and of saving a core image of lcs (named `foo` here) capturing that name. Knowing the name, the clients will be able to retrieve the service registered under it. Type safety is guaranteed by the fact that names and services are typed and that they have weak polymorphism; a name of type `ty name` may only be used to register or retrieve services of type `ty`. The registration primitives require the service registration daemon `lcsd` installed (see the Release Notes for the installation procedure).

```
unix1% lcs
val it = () : unit

- needs "lnetwork";

- val n = name () : (int -> int) service name;
val n = - : (int -> int) service name

- save "foo";
val it = () : unit
                                    unix%2 foo
                                    val it = () : unit
- prompt ("\n1> ","1= ");
val it = () : unit
                                    - prompt ("\n2> ","2= ");
                                    val it = () : unit
```

```
1> val k = open_pub_service
1= ("tippi.laas.fr",4501,"hello")
1= : (int -> int) service;

1> register_service (n,k);
val it = () : unit
                                      2> val k = get_service n;
                                      val k = - : (int -> int) service
1> val s = accept_connection k;
                                      2> val s = open_connection k;
                                      val s = - : (int -> int) stream
val s = - : (int -> int) stream

1> needs "bench";
1> export (s,Bench.fact);
val it = () : unit
                                      2> import s 5;
                                      val it = 120 : int

1> ^D                                 2> ^D
unix1%                                unix%2
```

SEE ALSO

Structures `Star`, `Ptop`, `Ring`, make intensive use of the `Remote` structure. They implement various network abstractions, with behavioral interfaces, from the above primitives.

Rstore

CONTENTS

Utilities for building and using remote storage daemons.

SIGNATURE

```
structure Rstore =
   structure Name =
      abstype 'a name
      val name        : unit -> '1a name
      val concat_name : 'a name -> string -> 'a name
      val equal_name  : 'a name * 'a name -> bool
   structure Sys =
      type rstore
      abstype token
      val t_open   : real -> rstore service -> token
      val t_cancel : token -> unit
      val t_commit : token -> unit
      val t_get          : token -> 'a name -> 'a
      val t_set                : token -> 'a name * 'a -> unit
      val t_unset              : token -> 'a name -> unit
      val t_update       : token -> 'a name * 'a -> unit
   structure Client =
      val get_store        : rstore service -> 'a name -> 'a
```

```
      val set_store        : rstore service -> 'a name * 'a -> unit
      val unset_store      : rstore service -> 'a name -> unit
      val update_store     : rstore service -> 'a name * 'a -> unit
   structure Server =
      val server : string * int * string -> _
```

DESCRIPTION

Structure `Rstore` provides generic facilities for creating typed databases that can be shared by lcs applications. It is used in particular to implement the service registration facilities of structure `Register`. It basically implements a store, with values referenced through names, and access and update primitives for that store. Stores may be run as daemons or may be part of applications.

Names:

`'a name`	Rstore values are registered under typed names. Type `ty name` is the type of references to values of type `ty`. Names do not admit equality, but a specific `equal_name` primitive is provided to compare them.
`name () : ty name`	Generates a new typed reference, for storing values of type `ty`. Each call of name is guaranteed to return a worldwide unique reference (well, almost, the probability of generating two equal names in two separate calls is indeed infinitesimal, though).
`concat_name n ext`	Creates a name by appending extension `ext` to the existing name `n`.
`Equal_name (n1,n2)`	Checks equality of names.

Remote storage services:

Access and update of remote storage. In case of failure, these routines raise exception `Remote` with a string parameter indicating the reason of failure.

`get_store r n`	Returns data registered under name `n` on rstore r, if any, otherwise fails.
`set_store r (n,v)`	Stores data v under name `n` on rstore r. Fails if name `r` is already in use in rstore `r`.
`update_store r (n,v)`	Replaces by v the data registered under name `n` on rstore r, if any, behaves like `set_store` otherwise.
`unset_store r n`	Removes data registered under name `n` on rstore r.

Registration server

`server (h,p,w)`	Behavior of an rstore server with connection parameters `(h,p,w)` (for host, pot number, and password, respectively).

System interface:

This interface is made available to allow one to implement specific remote storage protocols. These primitives are used, for instance, in network abstractions `Star` and `Ptop` to prevent concurrent activation of several networks with the same name. They may be used also to implement multi-site registration for lcs services. Clients perform transactions on the remote store daemon, transactions are ultimately committed or aborted. The server serializes transactions.

`rstore`	`rstore service` is the type conventionally given to rstore's connection services.
`token`	Type identifying an ongoing transaction between a client and an rstore server.

`t_open d r`	Opens a transaction with rstore server identified by service `r`. The transaction is aborted by the server if not completed in time `d`.
`t_cancel tk`	Cancels the transaction identified by token `tk`.
`t_commit tk`	Commits the transaction identified by token `tk`, the changes realized in the transaction become visible to other clients.
`t_get tk n`	Returns the value currently registered under name `n`, in transaction `tk`.
`t_set tk (n,v)`	Registers value v under name n, in transaction `tk`.
`t_unset tk n`	`Unr`egisters any value registered under name n, in transaction `tk`.
`t_update tk (n,v)`	Replaces value registered under n by v, in transaction tk.

EXAMPLES

Note: applications needed to share values MUST share the names under which these values are registered. This can be done by pre-allocating all names to be used in the application (as below, in which application mylcs captures the needed names), or by preallocating a single name used to retrieve other names. Function `concat_name` may also be helpful to reduce the number of names pre-allocated.

Creating a rstore on host tippi.laas.fr, port number 4502, with password "hi!":

```
unix1% lcs
- needs "lnetwork";
val it = () : unit
- reboot Rstore.Server.server ("tippi.laas.fr",4502,"hi!");
rstore server started
^Z
unix1% bg
```

Creating an application that captures some names:

```
unix2% lcs
- needs "lnetwork";
val it = () : unit
- val i = name () : int name;
val i = - : int name
- val r = name () : real name;
val r = - : real name
- save "mylcs";
val it = () : unit
- ^D
unix2%
```

Using that rstore though application `mylcs`, from any machine:

```
unix2% mylcs
- open Rstore.Client;
- val rs = get_pub_service ("tippi.laas.fr",4502,"hi!") : rstore service;
val rs = - : rstore service
- set_store rs (i,123);
val it = () : unit
- set_store rs (concat_name i "2",456);
val it = () : unit
- set_store rs (r,3.4E56);
```

```
val it = () : unit
- ^D
unix%2

unix%3 mylcs
- open Rstore.Client;
- val rs = get_pub_service ("tippi.laas.fr",4502,"hi!") : rstore service;
val rs = - : rstore service
- get_store rs i;
val it = 123 : int
- get_store rs (concat_name i "2");
val it = 456 : int
- get_store rs r;
val it = 3.4E56 : real
- ^D
unix%3
```

SEE ALSO

Ring

CONTENTS

Network abstraction; ring architecture.

SIGNATURE

```
structure ring =
   (* TO BE PROVIDED *)
```

DESCRIPTION

Network abstractions provide a behavioral abstraction for communication networks. Sites are numbered as they join the network, they may enter and leave the network at any time, and may exchange data with the other sites connected. Network abstractions `Star`, `Ptop`, and `Ring` provide the same interface but differ by their architecture. In the `Ptop` abstraction, the network server helps establish direct point-to-point links between each pair of clients, used to transfer site-to-site traffic.

`network` n	That behavior creates a network end (a client), with ports go, ind, req, and quit. The network is identified by name n. If the network is not registered yet, then it opens a server for the network, and then a client, both running in the same lcs machine, and registers the network. Otherwise, it simply opens a client and connects it to the existing network.
`'a group_msg`	The names used for creating networks must have type `ty group_msg name`, where `ty` is the type of data to be passed between sites; for type safety, type `ty` must be a monotype.

Network operation: Behavior `network n` first offers a client identifier on port `go` (an integer). Reading this identifier is mandatory, it allows the client's behavior to start. A client is terminated by sending it value `()` on port `quit`, the client acknowledges termination by emitting signal `TERM`. Data is sent to other clients connected to the network through port `req`, with a specified destination (see structure `Dest` for available destinations); data is received from other sites on port `ind`, with the identifier of the sender. Data transferred through the network must be monomorphic, this is essential for enforcing type safety.

EXAMPLES

SEE ALSO

Simple

CONTENTS

Simple broadcast network abstraction.

SIGNATURE

```
structure Simple =
   type 'a sname
   val simple : '1a sname -> {inp: '1a, out: '1a}_b
```

DESCRIPTION

Simple broadcast network abstraction. No site numbers; destinations are all other sites than emitter.

`'a sname`	Type to be given to the name identifying the network.
`simple n`	Simple network behavior. Produces a behavior that echoes to all other sites what is read on port `inp`. Data from other sites is offered on port `out`. Does not terminate (the lcs process must be killed). Server closes (signals `TERM`) when there are no more clients;

EXAMPLES

The following sequence of declarations generates a standalone application called `wecho` that opens a window and echoes in it what other current users of the same application write in their respective windows. Application `wecho` is terminated by typing `^C` in its window.

```
needs "simple";
val n = Register.Name.name () : string Simple.sname;
fun echo n w =
   let event BYE
       agent watchin = out?s => do (output_win (w,s)) => watchin
       agent watchout = inp! (input_line_win w) => watchout
       agent watchterm = signal (wait_ctrl_win w; BYE)
   in (Simple.simple n /\ watchin /\ watchout /\ watchterm) {/inp out}
      catch BYE => stop
   end;
reboot do (dump ("wecho","-q")) => echo n (open_win ());
```

SEE ALSO

`chat` demo.

Socket

CONTENTS

TCP sockets interface, with raw remote connections services.

SIGNATURE

```
structure Socket =
   exception Remote : string -> exn
   val open_service_  : string * int -> Dev.dev * string * int
   val close_service_ : Dev.dev * string * int -> unit
   val accept_connection_ : Dev.dev * string * int -> Dev.dev
   val open_connection_   : Dev.dev * string * int -> Dev.dev
   val close_connection_  : Dev.dev -> unit
   type key
```

DESCRIPTION

Bare interface to unix TCP sockets (connected mode only), and unsecured primitives for opening connection services and streams.

`Remote`	This exception is raised by the primitives below. Parameter tells the reason of failure.
`open_service_ (h,p)`	Given a host name `h` (or default `""`) and a port number `p` (or default 0), returns a triple constituted of an lcs device (encoding a TCP socket), the name of the host, and the port number allocated. Fails if host is not the local host.
`close_service_ dhp`	The arguments are typically those returned by a call of `open_service_`. Closes the socket encoded by the first component of `dhp`.
`accept_connection_ dhp`	Accepts a connection on the socket represented by the first component of `dhp`. Returns a device that encodes a bidirectional stream.
`open_connection_ dhp`	Opens a remote connection on the socket represented by the first component of `dhp`. Returns a device that encodes a bidirectional stream.
`close_connection_ d`	Closes lcs device d (assumed to represent a bidirectional stream).
`key`	A type abbreviation. A key groups together the information characterizing a TCP connection service (socket,host, port), and a password. This type is used in structure `Remote`.

EXAMPLES

(see usage in e.g. `Remote` structure)

SEE ALSO

Star

CONTENTS

Network abstraction; star architecture.

SIGNATURE

```
structure Star =
   type 'a group_msg
   val network : '1a group_msg name ->
                {go: int,                     (* go indication        *)
                 quit,                        (* quit request         *)
                 ind: int * '1a,              (* data indication      *)
                 req: dest * '1a}_b           (* data request         *)
```

DESCRIPTION

Network abstractions provide a behavioral abstraction for communication networks. Sites are numbered as they join the network, they may enter and leave the network at any time, and may exchange data with the other sites connected. Network abstractions `Star`, `Ptop`, and `Ring` provide the same interface but differ by their architecture. In the `star` abstraction, as implied by its name, all site-to-site communications transit through the network server.

`network` n	That behavior creates a network end (a client), with ports go, ind, req, and quit. The network is identified by name n. If the network is not registered yet, then it opens a server for the network, and then a client, both running in the same lcs machine, and registers the network. Otherwise, it simply opens a client and connects it to the existing network.
`'a group_msg`	The names used for creating networks must have type `ty group_msg name`, where `ty` is the type of data to be passed between sites; for type safety, type `ty` must be a monotype.

Network operation: Behavior `network n` first offers a client identifier on port `go` (an integer). Reading this identifier is mandatory, it allows the client's behavior to start. A client is terminated by sending it value `()` on port `quit`, the client acknowledges termination by emitting signal `TERM`. Data is sent to other clients connected to the network through port `req`, with a specified destination (see structure `Dest` for available destinations); data is received from other sites on port `ind`, with the identifier of the sender. Data transferred through the network must be monomorphic, this is essential for enforcing type safety.

EXAMPLES

See `chat` demo, or `Simple` structure.

SEE ALSO

3 scripts

These scripts do not define structures, but load and open a number of them. In addition, they may affect fixity or overloading status of some identifiers (what can't be achieved by opening structures).

lprelude

CONTENTS

Initial bindings.

DESCRIPTION

Installs the initial environment from the Core structure. Loads structure `Prelude` and installs it: opens all of structures in `Prelude` them except `Dev`, `Object`, `Array`, `Bytearray`,and `Vector`, overloading symbols `~ + - * div mod divide < > <= >= abs min max pow print makestring`, as well as `^^` when process option is not -n.

In addition to =, identifiers `* div / pow + - ^ :: @ <> < > <= >= o :=` are made infix. No identifier retains overloading status once installation is complete.

IMPLEMENTATION

```
read_file ( "prelude",false,false,true);  (* reloads *)

open Prelude;
infix 7 * div / pow;
infix 6 + - ^;
infixr 5 :: @;
infix 4 (* = *) <> < > <= >=;
infix 3 o :=;
infix ^^;
overload ~ + - * div mod divide < > <= >= abs min max pow print makestring;
open Exn;
open Unit;
open Pair;
open Bool;
open Int;
open Real;
open Mp;
open String;
open Ref;
open List;
open Lift;
open Fun;
open Io;
open Stdio;
open System;
open Host;
(* open Vector; *)
(* open Array; *)
(* open Bytearray; *)
(* open Dev; *)
(* open Object; *)
(* next three omitted if process option -n *)
```

```
open Beh;
open Evt;
open Void;
nonoverload ~ + - * div mod divide < > <= >= abs min max pow print makestring;
```

SEE ALSO

lextras

CONTENTS

General purpose programming structures.

DESCRIPTION

Sets the following infix: `// over ::: eq`. No identifier retains overloading status once installation is complete.

IMPLEMENTATION

```
needs "extras";

open Extras;
infix 8 // over;
infixr 5 :::;
infix 4 eq;
overload ~ + - * / mod divide < > <= >= abs min max pow over eq print makestring;
open Rat;
open Mprat;
open Rat.Rep;  (* so that // produces a rat *)
nonoverload ~ + - * / mod divide < > <= >= abs min max pow over eq print makestring;
open Alist;
open Slist;
open Tlist;
open Lizt;
open Stack;
open Stree;
(* open Avltree; *)
open Queue;
open Time;
(* open Print *)
(* open Term; *)
(* open Termio; *)
(* open Bench; *)
(* open Codec; *)
(* open Sort; *)
(* next six omitted if process option is -n *)
open Timing;
open Broadcast;
open Dataflow;
open Parrays;
open Pio;
```

SEE ALSO

lnetwork

CONTENTS

Distributed programming facilities: streams, services, registration facilities, network abstractions.

DESCRIPTION

Most of these structures require daemon lcsd installed. All require process option -l or -s, indifferently.

IMPLEMENTATION

```
needs "network";

open Network;
open Remote;
open Register.Name;
open Register.Client;
open Dest;
open Star;
(* open Ptop; *)
(* open Ring; *)
(* open Mutex; *)
(* open Simple; *)
(* open Go; *)
(* open Move; *)
(* open Socket; *)
(* open Rstore; *)
```

SEE ALSO

4 Index of bindings

A

B

C

D

I

K

L

M

N

O

P

S

T

U

V

W