The library, volume 2

The library, volume 2

The second volume of the report on raz. The first, `doc/raz.md`, defines the language; this one defines the modules that come with it, and refers to the first's sections by number, as "section 6.8". Only the core binds every implementation (appendix A of the first volume); the rest is the standard library.

Modules written in raz, in the compiler's `lib` directory:

Module	What
`Rt`	ending the program, `exit` and `atExit`; `stop`, which ends it with a message; `trace`, the calls under way; what the compiler's checks call when they fail
`Sys`	system calls: `open`, `openMode`, `openRead`, `create`, `read`, `write`, `seek`, `position`, `truncate`, `sync`, `syncAll`, `lock`, `close`, `mmap`, `munmap`, `isTerminal`, `yield`, `exit`; files and directories, `rename`, `remove`, `removeDir`, `makeDir`, `chmod`, `symlink`, `readLink`, `info`, `linkInfo`, `infoOf`, `openDir`, `nextEntry`, `closeDir`; the terminal, `rawMode`, `restoreMode`, `termSize`, `ready`; the environment, `env`, `workDir`, `setWorkDir`; programs, pipes and signals; time, `clock`, `now`, `sleep`; `random`; sockets, among them datagrams served and files passed; and memory shared through a file. One file per target, and `Sys.raz` over the C library for the others
`Out`	writing text and numbers to standard output, buffered; safe to call from several threads
`C`	the parts of the C library the others use, and `printf`, `fflush`, `stdout`, `pthread_create`; a program that imports it uses C
`Lock`	a lock for short sections, for C's threads and `Thread`'s alike
`Thread`	threads, `spawn` and `join`, and `Mutex`, in any program; in one that uses C, made by C's `pthread_create`
`Mem`	`Allocator`, the interface of every allocator, and pages from the system
`Heap`	a general allocator, `Heap.heap`, safe to call from several threads; freeing a block twice, or one it never gave out, stops the program
`Arena`	an allocator that frees everything at once; `ADR(arena.base)` is its allocator
`Str`	operations on `[]U8` strings: comparing, hashing, finding, trimming and cutting, case, joining and replacing, UTF-8, numbers to and from text
`Bytes`	searches of bytes, many at a time: `index`, `last`, `count` and `find` in a `[]U8`; one file per target where the target has instructions for it

Module	What
`Buf`	a growable byte buffer
`Fmt`	formatted output: `Fmt.print("{} of {}\n", [Fmt.int(n), Fmt.real(x, 2)])`, or into a `Buf` with `Fmt.write`
`List`	a generic singly linked list, `List(T)`
`Map`	a generic hash map, `Map(K, V; hash, eq)`
`Vec`	a generic growable array, `Vec(T)`, indexed as a slice
`Sort`	sorting a slice in place, `Sort(T; less)`
`Math`	roots, powers, logarithms, rounding and trigonometry, and `pi`, for `F64`
`Files`	an argument as a string, and a whole file read or written
`Opt`	a command line read through one table, by `getopt`'s rules with long options; its help and its mistakes

Lengths, counts, file descriptors, process ids and statuses are `INT`s, and a character is an `I32`. What may pass an `INT` on a small target is an `I64`: a file's size and position, a time in nanoseconds, and a number read from text.

What follows is each module's interface, as `tools/Doc.raz` prints it from the module's file; `...` stands for fields that are the module's own. A module's `main`, where it has one, is its test, and is left out here.

```
module Rt;
const Full*: ERROR = "too many procedures to run at exit";
proc atExit*(p: proc): ERROR;   (* p run when the program ends by exit *)
proc exit*(status: INT);        (* the atExit procedures, the last first, then the end
*)
proc stop*(msg: []U8);          (* msg on standard error, then exit with status 2 *)
proc trace*;                    (* the calls under way, and their variables, on
standard error *)
```

`Rt` also holds what the compiler's checks call when they fail, and the handlers of the signals that stop a program (section 10). `exit` takes each procedure off the list before it runs it, so it runs once even if it fails or calls `exit` itself; in a program that uses C, it ends through C's `exit`, so that C flushes its buffers too.

```
module Mem;
type
    Allocator* = record
        alloc*: proc(a: ^Allocator; size, align: INT): ^U8;   (* nil if out of room *)
        free*:  proc(a: ^Allocator; p: ^U8);
    end;
const Page* = 4096;
proc pages*(size: INT): ^U8;         (* size bytes, in whole pages, zeroed; or nil *)
proc unmap*(p: ^U8; size: INT);      (* pages given back *)
proc alloc*(a: ^Allocator; size, align: INT): ^U8;   (* a^.alloc(a, size, align) *)
proc free*(a: ^Allocator; p: ^U8);   (* a^.free(a, p) *)

module Heap;
var heap*: ^Mem.Allocator;

module Arena;
type Arena* = record
    base*: Mem.Allocator;     (* first, so that ^Arena(a) finds the rest *)
    ...
end;
proc init*(ar: ^Arena; size: INT): BOOL;   (* an arena of size bytes; false if the
system has none *)
proc reset*(ar: ^Arena);                   (* every allocation forgotten; the memory
used again *)
proc done*(ar: ^Arena);                    (* its memory given back *)
```

`NEW(p, a)` calls `a^.alloc`. `Heap` is the general allocator. An arena hands out memory from one block: its `free` does nothing, and `reset` forgets everything.

```
module Lock;
type Lock* = record ... end;
proc take*(l: ^Lock);
proc give*(l: ^Lock);

module Thread;
const Failed*: ERROR = "can't start a thread";
type
    Thread* = record ... end;
    Mutex* = record ... end;     (* zero is free *)
proc spawn*(p: proc(arg: ^U8); arg: ^U8): (^Thread, ERROR);   (* p(arg) in a new
thread *)
proc join*(t: ^Thread);         (* waits for t to end, then frees its stack *)
proc lock*(m: ^Mutex);
proc unlock*(m: ^Mutex);
```

A `Lock` is for short sections: a thread that finds it taken tries again, giving up the processor now and then. A `Mutex` sleeps. Each of `Thread`’s threads runs on a stack of its own, with a page below it that can’t be touched, so that running off the stack stops the program with a message, as in the main thread. In a program that uses C, `Thread` makes its threads with C’s `pthread_create`, so that C’s library knows of them; in one that doesn’t, with the system’s own calls. The program is the same either way.

```
module Out;
proc str*(s: []U8);
proc char*(c: U8);
proc ln*;                          (* a line feed *)
proc int*(n: I64);                 (* n in decimal *)
proc hex*(n: U64; digits: INT);  (* n in hexadecimal, digits digits at least *)
proc flush*;
```

Out and C's stdio. `Out` keeps a buffer of `Sys.outSize` bytes, 4 KB on Linux. It writes it out when it is full, at each line feed when standard output is a terminal, at `Out.flush`, and when the program ends by `Rt.exit`. A lock keeps what each call writes together when threads write at once. C's `stdio` keeps a buffer of its own for the same standard output. A program that writes with both flushes one before writing with the other, or the text comes out of order:

```
module Both;
import Out, C;      (* C: printf, fflush and C's FILE *stdout *)

proc main*;
begin
    Out.str("from raz\n");
    Out.flush;
    C.printf("from C\n");
    C.fflush(C.stdout);
    Out.str("from raz again\n");
end;

end Both.
```

Sys gives every target the same procedures, so that a program that needs only these runs wherever raz does: through system calls on Linux, through the C library elsewhere. A result that can fail is a number, minus an errno, as on Linux; 0 is success.

```
module Sys;
const
    stdin* = 0; stdout* = 1; stderr* = 2;
    outSize* = 4096;    (* the bytes Out keeps; each target has its own *)
    (* signals, by their numbers on Linux; each target has its own *)
    sigHup* = 1; sigInt* = 2; sigQuit* = 3; sigKill* = 9; sigPipe* = 13;
    sigTerm* = 15; sigChld* = 17; sigCont* = 18; sigStop* = 19;
    sigTstp* = 20; sigTtin* = 21; sigTtou* = 22; sigWinch* = 28;

type
    Access* = (reading, writing, creating, exclusive, truncating, appending);
    Accesses* = set of Access;
    Info* = record
        size*: I64;      (* in bytes *)
        mtime*: I64;     (* when its contents last changed, in nanoseconds since 1970
*)
        mode*: INT;      (* its permission bits, as chmod takes them: 1EDH is rwxr-xr-
x *)
        dir*: BOOL;      (* a directory *)
        file*: BOOL;     (* a regular file: not a directory, device, pipe or socket *)
        symlink*: BOOL;  (* the path itself is a symbolic link *)
        ino*: U64;       (* its inode, which a rename keeps and a new file doesn't *)
    end;
    Dir* = record ... end;   (* an open directory; its fields are the target's *)
    Program* = record
        path*: []U8;
        args*: [][]U8;      (* the first is its name *)
        env*: [][]U8;       (* NAME=value, each added to this program's environment *)
        onlyEnv*: BOOL;     (* env is its whole environment *)
        dir*: []U8;         (* its working directory; "" for this program's *)
        fds*: [3]INT;       (* its standard input, output and error; 0 for this
program's own *)
        group*: INT;        (* the process group it joins: 0 for this program's, -1 a
new one it leads *)
        foreground*: BOOL;  (* its group takes the terminal on standard input *)
        tty*: INT;          (* a pseudo-terminal's fd, to start a session on; 0 for
none *)
    end;
    Status* = (running, exited, killed, stopped);
    Peer* = record ... end;   (* whom a datagram came from *)

proc write*(fd: INT; s: []U8): INT;     (* the bytes written *)
proc read*(fd: INT; buf: []U8): INT;    (* the bytes read; 0 at the end *)
proc open*(path: []U8; how: Accesses): INT;   (* its fd *)
proc openMode*(path: []U8; how: Accesses; mode: INT): INT;   (* open, a file it makes
given mode *)
proc openRead*(path: []U8): INT;         (* open(path, {reading}) *)
proc create*(path: []U8): INT;           (* open(path, {writing, creating,
truncating}) *)
proc seek*(fd: INT; pos: I64; fromEnd: BOOL): I64;  (* the new position *)
proc position*(fd: INT): I64;            (* where the next read or write is *)
proc truncate*(fd: INT; size: I64): INT; (* the file made size bytes long *)
proc sync*(fd: INT): INT;                (* what was written to it, on the disk *)
proc syncAll*(fd: INT): INT;             (* all that was written to its file system,
on the disk *)
```

```
proc lock*(fd: INT; wait: BOOL): INT;    (* the file's lock, held until fd is closed
*)
proc close*(fd: INT);
proc isTerminal*(fd: INT): BOOL;
proc mmap*(size: INT): ^U8;              (* size bytes of zeroed memory, in whole
pages; or nil *)
proc munmap*(p: ^U8; size: INT);
proc yield*;                             (* the processor given to another thread *)
proc exit*(status: INT);                 (* the program ended at once, without Rt's
atExit *)

proc rename*(old, new: []U8): INT;       (* replaces what new names *)
proc remove*(path: []U8): INT;           (* a file, not a directory *)
proc removeDir*(path: []U8): INT;        (* an empty directory *)
proc makeDir*(path: []U8): INT;          (* readable by all, writable by its owner *)
proc chmod*(path: []U8; mode: INT): INT;
proc symlink*(target, path: []U8): INT;  (* path made a link to target *)
proc readLink*(path: []U8): ([]U8, INT); (* what the link names, good until the next
call *)
proc info*(path: []U8): (Info, INT);     (* following a symbolic link *)
proc linkInfo*(path: []U8): (Info, INT); (* the link itself *)
proc infoOf*(fd: INT): (Info, INT);      (* the open file's *)
proc space*(path: []U8): (I64, I64, INT);   (* the bytes free and in all on path's
file system *)

proc openDir*(d: ^Dir; path: []U8): INT;
proc nextEntry*(d: ^Dir): ([]U8, BOOL);  (* a name, good until the next call; false at
the end *)
proc closeDir*(d: ^Dir);

proc rawMode*(fd: INT): BOOL;      (* the terminal on fd: no echo, no lines, no
signals *)
proc restoreMode*(fd: INT);        (* as rawMode found it *)
proc termSize*(fd: INT): (INT, INT);   (* columns and rows; 80 and 24 if unknown *)
proc ready*(fd, ms: INT): BOOL;          (* something to read within ms, or ever if ms
< 0 *)
proc readyOf*(fds: []INT; ms: INT): INT;   (* the first of fds with something to read
within ms, or -1 *)
proc readyFor*(reads, writes: []INT; ms: INT): INT;   (* readyOf, and the first of
writes with room to write *)
proc setBlocking*(fd: INT; yes: BOOL): INT;   (* if not yes, a read or write that
would wait is -11 at once *)

proc env*(name: []U8): []U8;       (* the environment variable's value; "" if unset *)
proc workDir*: ([]U8, INT);        (* the working directory, good until the next call
*)
proc setWorkDir*(path: []U8): INT;

proc pipe*: (INT, INT);            (* the fd to read and the fd to write; or minus an
errno and -1 *)
proc openPty*: INT;                (* a pseudo-terminal: the fd of its controlling
side *)
proc setTermSize*(fd, cols, rows: INT): INT;   (* the size its programs are told of *)
proc spawn*(p: Program): INT;      (* a program started: its process id *)
proc wait*(pid: INT; block: BOOL): (INT, Status, INT);   (* which program, how, and
its code *)
```

```
proc kill*(pid, signal: INT): INT;
proc pid*: INT;                     (* this program's process id *)
proc setGroup*(pid, group: INT): INT;    (* pid 0 is this program; group 0 a new one
pid leads *)
proc foreground*(fd, group: INT): INT;   (* the terminal on fd gives its keys to group
*)
proc foregroundOf*(fd: INT): INT;  (* the group the terminal on fd gives its keys to
*)

proc catch*(sigs: []INT): INT;     (* sigs caught: an fd to wait on *)
proc caught*(fd: INT): INT;        (* the next signal caught, or 0 if none is waiting
*)
proc ignore*(sig: INT; yes: BOOL): INT;  (* sig ignored; or, if not yes, back to what
it does at first *)

proc clock*: I64;                  (* nanoseconds, on a clock that only goes forward
*)
proc now*: I64;                    (* nanoseconds since 1970-01-01, UTC *)
proc sleep*(ms: INT);
proc random*(b: []U8);             (* b filled with random bytes *)

proc connect*(path: []U8): INT;    (* a stream socket, connected to the Unix socket at
path: its fd *)
proc connectTcp*(addr: []U8; port: INT): INT;   (* addr an IPv4 or IPv6 address, as
numbers *)
proc connectUdp*(addr: []U8; port: INT): INT;
proc listen*(path: []U8): INT;     (* a Unix socket made at path, taking connections
*)
proc listenTcp*(addr: []U8; port: INT): INT;
proc accept*(fd: INT): INT;        (* the next connection, waiting for one *)
proc send*(fd: INT; s: []U8): INT; (* write, to a socket: -32 (EPIPE), not a signal,
if it is closed *)
proc connectTcpWithin*(addr: []U8; port, ms: INT): INT;   (* connectTcp, or -110
(ETIMEDOUT) after ms *)
proc listenUdp*(addr: []U8; port: INT): INT;   (* a socket of datagrams on port at
addr *)
proc receive*(fd: INT; buf: []U8): (INT, Peer);   (* the next datagram, in buf, and
who sent it *)
proc reply*(fd: INT; peer: Peer; s: []U8): INT;   (* s, as one datagram, to peer *)
proc localPort*(fd: INT): INT;     (* the port the socket fd is bound to *)
proc sendFds*(fd: INT; s: []U8; fds: []INT): INT;   (* send, and with it fds, to a
Unix socket *)
proc receiveFds*(fd: INT; buf: []U8; fds: []INT): (INT, INT);   (* read, and the fds
that came with it *)

proc memFile*(size: INT): INT;     (* a file of size zero bytes, in memory, with no
name: its fd *)
proc mapFile*(fd, size: INT): ^U8; (* the first size bytes of fd, shared; munmap gives
them back; or nil *)
```

`info` follows a symbolic link: `size`, `dir` and `file` are those of what it names, and `symlink` says the path was a link, so that a program walking a tree can choose not to follow one back up it. A link to nothing is `-ENOENT`, as a missing file is. `file` lets a walk open only what holds data: opening a pipe would wait for a program to write to it.

`nextEntry` gives every name in the directory except `.` and `..`, in no particular order. `rawMode` turns off what the terminal does with keys before the program reads them (echo, line editing, signals, flow control), and NL to CR NL on output. A program that turns it on restores it before it ends: `Rt.atExit` is the place.

Files. `open` gives a file for what `how` asks: `reading`, `writing` or both; `creating` it if it is missing, readable by all and writable by its owner; with `exclusive`, failing with -17 (EEXIST) if it is there, as a program does that makes a lock or a file of its own; `truncating` it to nothing; `appending`, each write going to its end, however other programs write it too. `openMode` is `open` that gives a file it makes the bits `mode`, less those the umask takes away: 1EDH (755) for a program, 124H (444) for a file never to be written; `open` gives 1A4H (644). `seek` moves to `pos` from the start of the file, or from its end if `fromEnd`, where a negative `pos` reads back from the end; `position` says where the file is, without moving. `truncate` makes the file `size` bytes long, cutting it or adding zeros, and leaves the position where it was. `sync` returns once what was written to the file is on the disk, as a program needs before it says a file is saved; `syncAll` returns once everything written to the file system that holds the file is on the disk, at once rather than a file at a time, and may flush other file systems too. `lock` takes the file's lock, which one open file holds at a time: another that asks for it waits if `wait`, and otherwise fails with -11 (EAGAIN). The lock goes when the file is closed or the program ends, however it ends, so it never goes stale as a file made with `exclusive` does. `infoOf` is `info` of an open file: its size is what it holds now, however it was written; `linkInfo` is `info` of a symbolic link itself, not what it names, so that `symlink` is true, `dir` and `file` false, and `size` the length of what it names, which need not be there. `Info.mtime` is what a program checks to see whether a file changed under it, and `Info.ino` whether it is still the same file, not one renamed over it; 0 on a target whose files have none. `Info.mode` holds the nine bits of `rwxrwxrwx`, and `chmod` sets them, 1EDH (755) for a program, 1A4H (644) for data.

Programs. `openPty` makes a pseudo-terminal, a terminal that a program, a terminal emulator, plays the screen and the keyboard of: what it writes to the fd is typed, and what the programs on the terminal write it reads.

`spawn` starts the program at `p.path`, with `p.args` as its arguments, the first being its name. Its environment is this program's with each `NAME=value` of `p.env` added or put in place of the old value, or, if `p.onlyEnv`, `p.env` alone. It starts in `p.dir`. Its standard input, output and error are `p.fds`; a 0 gives it this program's own, so `{path: "/bin/ls", args: ["ls"]}` runs `ls` as a shell runs it, with every other field zero. It gets no other open file, every signal does what it does at first, and none is blocked. If `p.tty` is a pseudo-terminal's fd, the program starts a session on that terminal: it is its controlling terminal, and its standard input, output and error, and `p.fds` and `p.group` are not used.

`p.group` puts the program in a process group, the programs that a shell's job is made of: -1 makes it the first of a new group, whose number is its process id, and the other programs of the job join that number. The terminal gives its keys, and the signals of Ctrl-C and Ctrl-Z, to one group, the one in its foreground: `p.foreground` puts the program's group there before it starts, and `foreground` puts any group there, as a shell does to take the terminal back. `setGroup` moves a program to a group, as the shell does for each program it starts, for whichever of the two runs first; a shell runs in a group of its own, `setGroup(0, 0)`.

`wait` waits for the program `pid` to change, or for any of this program's if `pid` is -1, and says which one, and how: `exited`, with its exit status; `killed`, with the number of the signal that ended it; or `stopped`, by Ctrl-Z or a signal, with the signal's number. Without `block` it doesn't wait, and is 0 and `running` if no program changed. A program that has nothing to wait for is -10 (ECHILD). `kill` sends a program a signal, `sigTerm` to end it; to a group, if `pid` is minus its number.

Signals. `catch` takes the signals `sigs` away from what they would do, ending the program most of them, and gives an fd that is ready to read when one comes, so that a program waits for signals with `ready` and `readyOf`

as it waits for keys or a server. `caught` gives each signal that came, in turn. A later `catch` adds to the signals caught and gives the same fd. Catching `sigKill` or `sigStop`, or the signals `Rt` reports, SIGSEGV and SIGFPE, is -22 (EINVAL). `ignore` makes a signal do nothing, as a shell ignores `sigTtou` so that it can take its terminal back, and `ignore(sig, false)` gives it back what it does at first. A program that `spawn` starts begins with neither.

Time. `clock` is for measuring how long something takes: it never goes back, and says nothing of the date. `now` is the date and time, which a person or the network may set back. `sleep` waits `ms` milliseconds.

Sockets. `connect` is for talking to a server on the same machine, an X server at `/tmp/.X11-unix/X0` say; `connectTcp` for one across the network, given its address as numbers, `"127.0.0.1"` or `"::1"`: turning a name into an address is a library's work over these. `connectUdp` gives a socket of datagrams, each `send` one message to that address and each `read` one message from it, as a program asks a name server. What a socket receives comes by `read`, what it sends goes by `send`, and `close` ends it. `send` is `write` for a socket, except when the other end has closed it: `write` would stop the program with the signal SIGPIPE, where `send` returns -32 (EPIPE), so that a program learns its server has gone and can end as it wants. `listen` and `listenTcp` are the other side: a server's socket, on `path`, which must not be there yet, or on `port` at `addr`, `"0.0.0.0"` or `"::"` for every address the machine has; `accept` gives each connection that comes, as a socket of its own. A server started on port 0 gets a port the system chooses, which `localPort` says, as a test does that must not take a port in use. `connectTcp` waits as long as the system does, minutes for a host that never answers; `connectTcpWithin` gives up after `ms` milliseconds, with -110 (ETIMEDOUT).

`listenUdp` is the server's side of `connectUdp`: a socket of datagrams from anyone, as a name server has. `receive` gives the next datagram, at most `LEN(buf)` bytes of it, the rest lost, and a `Peer` that `reply` sends an answer to.

Passing files and sharing memory. A program can give another an open file over a Unix socket: `sendFds` sends `s`, at least one byte, and with it `fds`, at most 16, each a file the other program then has open too; `receiveFds` reads into `buf` as `read` does, puts the fds that came with it in `fds`, as many as there is room for, closing the rest, and says how many bytes and how many fds. Each is closed in a program exec starts. `memFile` makes a file that lives in memory, of `size` zeroed bytes, and `mapFile` maps it, or any file, so that what one program writes there another sees at once: so a program hands an X server or a Wayland compositor its pixels, by `sendFds` of a `memFile` both map, rather than by copying them down a socket.

`readyOf` waits on several file descriptors at once, as a program does that listens to a server and a terminal both; a descriptor at its end, or failed, counts as ready, since a read of it won't wait. It looks at the first 64 of `fds`, and `connect` and `listen` take a path of at most 107 bytes, as `struct sockaddr_un` holds; a longer one is -36 (ENAMETOOLONG).

A program that both reads from a descriptor and writes much to it, as a terminal writes what is pasted to its program while it reads what the program writes back, must not wait on either: the program may be waiting for its own writes to be read. `setBlocking(fd, false)` has a read or a write on `fd` that would wait return -11 (EAGAIN) at once, a write having written what there was room for; `readyFor` waits, as `readyOf` does, for something to read on any of `reads` or for room to write on any of `writes`. A descriptor in both is given once, when it is ready for either; the 64 it looks at are `reads`, then `writes`.

Strings, buffers and collections.

```
module Str;
proc cmp*(a, b: []U8): INT;           (* -1, 0 or 1 as a sorts before, with or after
b, byte by byte *)
proc eq*(a, b: []U8): BOOL;
proc eqFold*(a, b: []U8): BOOL;       (* eq, an ASCII letter's capital and small the
same *)
proc copy*(a: ^Mem.Allocator; s: []U8): ([]U8, BOOL);   (* s's bytes in memory from a;
false if a has none *)
proc hash*(s: []U8): U64;             (* FNV-1a, 64 bits *)
proc find*(s, sub: []U8): INT;        (* where sub first starts in s, or -1 *)
proc findLast*(s, sub: []U8): INT;    (* where sub last starts in s, or -1 *)
proc hasPrefix*(s, p: []U8): BOOL;
proc hasSuffix*(s, x: []U8): BOOL;
proc trim*(s: []U8): []U8;            (* s without the spaces at either end *)
proc trimLeft*(s: []U8): []U8;        (* s without the spaces at its start *)
proc trimRight*(s: []U8): []U8;       (* s without the spaces at its end *)
proc cut*(s, sep: []U8): ([]U8, []U8, BOOL);   (* what comes before the first sep, and
after it *)
proc field*(s: []U8): ([]U8, []U8);   (* the first run of bytes in s that aren't
spaces, and what follows it *)
proc lower*(s: []U8);                 (* s's ASCII capitals made small, in place *)
proc upper*(s: []U8);                 (* s's ASCII small letters made capitals, in
place *)
proc join*(b: ^Buf.Buf; parts: [][]U8; sep: []U8);   (* the parts added to b, sep
between each two *)
proc replace*(b: ^Buf.Buf; s, old, new: []U8);       (* s added to b, with new for
each old in it *)
proc encode*(c: I32): ([4]U8, INT);   (* c in UTF-8: its bytes, and how many *)
proc decode*(s: []U8): (I32, INT);    (* the character s starts with, and its length
in bytes *)
proc decodeLast*(s: []U8): (I32, INT);   (* the character s ends with, and its length
in bytes *)
proc next*(s: []U8; i: INT): (I32, INT);  (* the character at i, and where the next
starts; -1 at the end *)
proc valid*(s: []U8): BOOL;           (* s is UTF-8, every character rightly spelled
*)
proc chars*(s: []U8): INT;            (* the characters in s, as decode reads them *)
proc toInt*(s: []U8): (I64, BOOL);    (* s, in decimal; false if it isn't a number an
I64 holds *)
proc toHex*(s: []U8): (U64, BOOL);    (* s, in hexadecimal; false if it isn't a number
a U64 holds *)
proc fromInt*(n: I64): ([20]U8, INT); (* n in decimal: its bytes, and how many *)
proc le*(s: []U8): U64;               (* s, at most 8 bytes, the first the lowest *)
proc be*(s: []U8): U64;               (* s, at most 8 bytes, the first the highest *)
proc putLE*(s: []U8; v: U64);         (* v into s, the lowest byte first *)
proc putBE*(s: []U8; v: U64);         (* v into s, the highest byte first *)

module Buf;
type Buf* = record len*: INT; ... end;
proc init*(b: ^Buf; a: ^Mem.Allocator);
proc room*(b: ^Buf; n: INT): BOOL;    (* room for n more bytes; false if the allocator
has none *)
proc add*(b: ^Buf; c: U8);
proc str*(b: ^Buf; s: []U8);
proc spare*(b: ^Buf; n: INT): []U8;   (* room for n more bytes, and those bytes, to
```

```
fill *)
proc bytes*(b: ^Buf): []U8;          (* what it holds, until it next grows *)
proc reset*(b: ^Buf);                (* emptied, its memory kept *)
proc done*(b: ^Buf);                 (* its memory given back *)

module Fmt;
type Arg* = variant
    int(i: I64);
    uint(u: U64);
    hex(h: U64);
    str(s: []U8);
    char(c: U8);
    bool(b: BOOL);
    real(r: F64; places: INT);
end;
proc print*(f: []U8; args: []Arg);              (* to standard output *)
proc write*(b: ^Buf.Buf; f: []U8; args: []Arg); (* added to b *)

module List(T);
type
    Node* = record next*: ^Node; val*: T; end;
    List* = record head*: ^Node; len*: INT; ... end;
proc init*(l: ^List; a: ^Mem.Allocator);
proc push*(l: ^List; v: T): BOOL;     (* v at the front; false if the allocator has no
room *)
proc pop*(l: ^List): (T, BOOL);       (* the front value, taken off; false if the list
is empty *)
proc reverse*(l: ^List);
proc clear*(l: ^List);

module Map(K, V; hash: proc(k: K): U64; eq: proc(a, b: K): BOOL);
type Map* = record len*: INT; ... end;
proc init*(m: ^Map; a: ^Mem.Allocator);
proc put*(m: ^Map; k: K; v: V): BOOL;         (* k now maps to v; false if there was
no room *)
proc remove*(m: ^Map; k: K): BOOL;            (* false if k wasn't there *)
proc get*(m: ^Map; k: K): (V, BOOL);          (* what k maps to, and whether it maps
to anything *)
proc next*(m: ^Map; i: INT): (INT, K, V);     (* the entry after i *)

module Vec(T);
type Vec* = record len*: INT; ... end;
proc init*(v: ^Vec; a: ^Mem.Allocator);
proc room*(v: ^Vec; n: INT): BOOL;    (* room for n more; false if the allocator has
none *)
proc push*(v: ^Vec; x: T);            (* x added at the end *)
proc add*(v: ^Vec): ^T;               (* a new element at the end, zero, to fill in
place *)
proc resize*(v: ^Vec; n: INT);        (* n elements: cut short, or the new ones zero
*)
proc all*(v: ^Vec): []T;              (* the elements, until it next grows *)
proc reset*(v: ^Vec);                 (* emptied, its memory kept *)
proc done*(v: ^Vec);                  (* its memory given back *)

module Sort(T; less: proc(a, b: T): BOOL);
proc sort*(s: []T);
```

```
module Math;
const pi* = 3.141592653589793;
proc sqrt*(x: F64): F64;
proc pow*(x, y: F64): F64;            (* x to the power y *)
proc floor*(x: F64): F64;
proc ceil*(x: F64): F64;
proc round*(x: F64): F64;             (* to the nearest whole number; a half away from
0 *)
proc exp*(x: F64): F64;
proc ln*(x: F64): F64;
proc log2*(x: F64): F64;
proc log10*(x: F64): F64;
proc sin*(x: F64): F64;
proc cos*(x: F64): F64;
proc tan*(x: F64): F64;
proc asin*(x: F64): F64;              (* from -pi/2 to pi/2 *)
proc acos*(x: F64): F64;              (* from 0 to pi *)
proc atan*(x: F64): F64;              (* from -pi/2 to pi/2 *)
proc atan2*(y, x: F64): F64;          (* the angle of (x, y) from the x axis, from -pi
to pi *)
proc bits*(x: F64): U64;              (* the bits of x, as IEEE 754 lays them out *)
proc fromBits*(u: U64): F64;          (* the F64 whose bits are u *)
proc mul*(a, b: U64): (U64, U64);     (* a times b: the low 64 bits of the product,
and the high *)
proc toReal*(s: []U8): (F64, BOOL);   (* s as a real, the F64 nearest it; false if it
isn't one *)
```

A `Buf`, a `List`, a `Map` and a `Vec` take their memory from the allocator `init` gives them. `Buf.add`, `str` and `spare` stop the program, as a failed check does, if the allocator has no room; `room` asks first. A program reads into a `Buf` through `spare`, then adds what it read to `len`: `s := Buf.spare(b, 4096); n := Sys.read(fd, s); if n > 0 then INC(b^.len, n); end;`. A `Vec` is `Buf` for any `T`: one block, which doubles when it is full, so `push` costs the same on average however long it grows. `all` gives the elements as a slice, indexed and checked as any is, and `v.len` is their number; the slice, and what `add` gives, are good until the array next grows. `push`, `add` and `resize` stop the program, as `Buf.add` does, if the allocator has no room; `room` asks first: `import Lines := Vec([]U8);`. `Fmt`'s format takes the next argument at each `{}`; `{{` and `}}` stand for `{` and `}`. A `Map` hashes and compares its keys with the procedures it is given: `import Count := Map([]U8, INT, Str.hash, Str.eq);`.

`Str` reads and writes UTF-8. `encode` writes a character, of at most 10FFFFH, in one to four bytes; a number that isn't one becomes FFFDH, the character that stands for a wrong one. `decode` reads the first character of `s`, and gives FFFDH and 1 for a byte that doesn't start one rightly, and 0 and 0 for an empty `s`: a continuation, a byte followed by too few continuations, a surrogate, a character beyond 10FFFFH or one spelled in more bytes than it needs. `decodeLast` reads the last character in the same way, to walk a string backwards. `valid` says that `decode` finds no such byte in `s`, and `chars` counts the characters, each such byte one. `next` walks a string a character at a time, as `decode` reads them: from 0, it gives the character at `i` and where the one after it starts, and 0 and -1 when `i` is at the end, or past it:

```
c, i := Str.next(s, 0);
while i >= 0 do
    ...                                 (* c is a character of s *)
    c, i := Str.next(s, i);
end;
```

A string is bytes, and `Str` takes no more of Unicode than UTF-8: `lower`, `upper` and `eqFold` change and match ASCII letters alone, and nothing says how wide a character is on a terminal, nor which spellings of a text are the same.

A space, to `trim`, `trimLeft`, `trimRight` and `field`, is a blank, a tab, a line feed, a carriage return, a vertical tab or a form feed. `cut` gives what comes before the first `sep` in `s`, what comes after it, and `true`; or `s`, an empty string and `false` if `s` has no `sep`. `field` passes the spaces at the start of `s` and gives the bytes up to the next space or the end, and the rest of `s` from there; two empty strings if `s` holds only spaces. They take a string apart without memory of their own, each part a slice of `s`:

```
line, rest, more := Str.cut(text, "\n");     (* the first line, and the others *)
key, val, ok := Str.cut(line, "=");
w, rest := Str.field(line);                 (* the words of a line *)
while LEN(w) > 0 do
    Out.str(w); Out.ln;
    w, rest := Str.field(rest);
end;
```

`lower` and `upper` write `s`'s bytes, so they take a variable's, not a string literal's (section 5.10). `join` and `replace` add to a `Buf` as `Buf.str` does; an empty `old` adds `s` as it is.

`toInt` reads a decimal number, with a `-` or `+` before it if any, and nothing else. `toHex` reads hexadecimal digits, capitals or not, and nothing else: no sign, `0x` or `H`. `fromInt` writes a number as `Out.int` does, with a `-` if it is negative, its bytes the first of the 20 it gives.

`le` and `be` read an unsigned number of up to 8 bytes in a fixed order, little-endian or big-endian, whatever the machine's own, and `putLE` and `putBE` write one; more than 8 bytes stops the program. They are how a file format or a network protocol is read and written:

```
len := U32(Str.le(header[4..8]));
Str.putBE(packet[0..2], U64(port));
```

`Math.toReal` reads a real as a constant is written (section 2), with a `-` or `+` before it if any, its point and the digits after it, or its exponent, left out if it has none, and `e` taken as `E`: `2`, `-0.5` and `1.5e-3` are reals to it. It gives the `F64` nearest the number, and `false` for anything else, or a number too big for an `F64`.

`Map.next` walks a map: from 0, it gives the next entry and the number to pass for the one after it, and -1 when there are no more. A walk sees each entry once if the map doesn't change during it:

```
i, k, v := Names.next(ADR(names), 0);
while i >= 0 do
    Fmt.print("{} is {}\n", [Fmt.str(k), Fmt.int(v)]);
    i, k, v := Names.next(ADR(names), i);
end;
```

`Sort.sort` puts `s` in the order `less` gives, in place, without memory of its own, in time `n log n` at most; two elements equal by `less` may change places. `Math`'s procedures are as C's `math.h` has them, to within the last bit or so. `Fmt.real(x, places)` writes `x` with `places` digits after the point, at most 9, rounded to the nearest; from 1e15 on, with an exponent, as `1.50e17`; and `nan`, `inf` and `-inf` as themselves.

Files and Opt are what most programs need of their command line.

```
module Files;
proc arg*(i: INT): []U8;                     (* argument i; 0 is the program's name *)
proc read*(path: []U8; b: ^Buf.Buf): BOOL;   (* the whole file added to b *)
proc write*(path, data: []U8): BOOL;         (* the file made to hold data *)

module Opt;
type
    Row* = variant
        usage(scope, line: []U8);           (* a synopsis line, without "usage: " *)
        sub(scope, name, help: []U8);       (* a subcommand *)
        option(scope: []U8; letter: U8; long, value, help: []U8);
    end;
    Parser* = record
        letter*: U8;    (* the option just read *)
        value*: []U8;   (* its value; "" for a flag *)
        long*: BOOL;    (* it was written by its long name *)
        ...
    end;
proc init*(p: ^Parser; name: []U8; table: []Row);  (* the arguments from 1 on, by
table *)
proc next*(p: ^Parser): BOOL;                      (* the next option; false when they
end *)
proc operand*(p: ^Parser): ([]U8, BOOL);           (* the next operand; false when
there are no more *)
proc at*(p: ^Parser): INT;                         (* the argument operand would give
next *)
proc scope*(p: ^Parser; name: []U8);               (* subcommand name's options, from
the word after it *)
proc number*(p: ^Parser; lo, hi: I64): I64;        (* the value just read, a number
from lo to hi *)
proc fail*(p: ^Parser; text: []U8);                (* the program's own mistake *)
```

`Opt` reads a command line by the rules of the C library's `getopt`, with GNU's long options, so that a program's options behave as every other Unix program's. The program describes itself once, in a table of `Row`s: its synopsis lines, its subcommands and its options. The help and the messages for mistakes are made from the table, so they can't drift from it; the program's test of `p.letter` is the only other place an option appears. The parser is a record the program owns, and the table a slice it keeps as long as the parser: a literal written in the call to `init` lives as long as the procedure that makes it.

A row's `scope` is `""` for the program itself, or a subcommand's name: `"start"`, or `"remote.add"` for a subcommand of `remote`. An option has a `letter`, a `long` name, or both. One with a long name only has a key of its own below `'!'` for a letter, so that the program can tell it from the others. With a `value`, the name the help and the messages give it, the option takes a value; with `""` it is a flag.

```
const wait = 1;     (* the key of --wait, which has no letter *)
...
Opt.init(ADR(p), "keep", [
    Opt.usage("", "keep [-v] [--wait secs] start ..."),
    Opt.sub("", "start", "start a job"),
    Opt.option("", 'v', "verbose", "", "say more"),
    Opt.option("", wait, "wait", "secs", "give the job secs to start"),
    Opt.usage("start", "keep start [-b label] name command [argument ...]"),
    Opt.option("start", 'b', "bar", "label", "show the job in the bar as label")]);
while Opt.next(ADR(p)) do
    if p.letter = 'v' then verbose := true;
    elsif p.letter = wait then secs := Opt.number(ADR(p), 1, 600);
    end;
end;
w, ok := Opt.operand(ADR(p));
if not ok or (w # "start") then Opt.fail(ADR(p), "no command; see keep -h"); end;
Opt.scope(ADR(p), "start");
while Opt.next(ADR(p)) do
    if p.letter = 'b' then bar := p.value; end;
end;
job, ok := Opt.operand(ADR(p));
```

The rules are `getopt`'s:

- Flags may share a dash: `-lw` is `-l -w`. A letter's value is the rest of its word, `-bdev`, or the whole next word, `-b dev`, even one that starts with a dash or is empty; in a group, the first letter that takes a value takes the rest, `-qn12`.
- A long option is written in full, with its value after a `=` or in the next word: `--bar=dev`, `--bar dev`. An abbreviation is unknown, not guessed, so that adding an option never breaks a script. A value given to a flag, `--follow=x`, is a mistake.
- `--` ends the options, and isn't an operand; `-` alone is one. The options end at the first operand, so in `keep start site python -m http.server` the `-m` is python's.
- Each scope has its own options. An option given twice is read twice.

`number` reads the value as `Str.toInt` does. `operand` reads what options are left first, and the program doesn't see them. So does `at`, which says where the operands are: the arguments from `operand`'s next on are `Files.arg(i)` for `i` from `at(p)` to `ARGC() - 1`, so that an interpreter hands a script its arguments as they are:

```
script, ok := Opt.operand(ADR(p));       (* tery file arg ... *)
first := Opt.at(ADR(p));                 (* file's own arguments, from here on *)
```

`-h` or `--help` among a scope's options prints its help on standard output and ends the program with status 0. `init` and `scope` answer it, looking through the scope's options before `next` gives one, so that the program has done nothing yet: `keep -v -h -x` prints the help, and the program never sees `-v`, nor the mistake. The help is the scope's synopsis lines, a blank line, and its subcommands, its options and the help itself, each with its line of help:

```
usage: keep start [-b label] name command [argument ...]

  -b, --bar label   show the job in the bar as label
  -h, --help        print this and exit
```

A scope with several synopsis lines has the others indented under the first; one with none, `usage:` and its words, as `usage: keep start`.

A mistake on the command line is a sentence on standard error, the program's name first, then the scope's synopsis lines, as the help begins, and status 2:

```
keep: unknown option -x
usage: keep start [-b label] name command [argument ...]
```

The sentences are `unknown option -x` (or `--name`, up to its `=`), `-b needs a label`, `--follow takes no value` and `--wait wants a number from 1 to 600, not '0'`. `fail` writes the program's own mistakes the same way, but alone, so that the sentence can say where the help is: `keep: no command; see keep -h`.

A mistake in the table is the program's, not its user's. `init` checks the table before it reads a word, and ends the program with status 1: `keep: option table, row 3: the option is in the scope already`. The others are an option with neither a letter nor a long name, a long name with `=` in it, a letter `-`, and `-h` or `--help` in the table, since they are `Opt`'s.