FTL User's Guide

Introduction

Version

This guide applies to the FTL run-time library with version 1.27.

Overview

The primary use of FTL is to facilitate the simple creation of tools driven by command line scripts that can either be read from a script file or typed line-by-line on a console. This is achieved by writing programs in C or C++ which create FTL *commands* and FTL *functions* which are used directly to provide the basic operations a tool wishes to make available. This C or C++ can make use of the FTL library which

- can handle common aspects of a script interpretation
- provides a run-time library of additional FTL commands, functions and values which can be made available alongside the features provided by the tool.

The advantage to the tool writer of using FTL is that the simple script language supported can be extended using the FTL language in such a way that the scripts made available to the tool user can be far more sophisticated than would otherwise been the case.

The typical structure of a tool written using FTL involves

- the FTL run-time library
- zero or more FTL run-time library extensions
- C/C++ code written by the tool writer
- a "prolog" written in FTL which provides a user scripting environment based on that provided by the tool writer

The FTL language itself is syntactically more diverse (and therefore more complex for the tool user to learn) than the basic script syntax we'd expect tool users to require. Although FTL is available to the tool user, typically only the separate, and simpler, basic script syntax should be needed.

Basic Script Syntax

Command lines are extracted from a command file (or command line console) after

- discard of commented-out lines (those beginning '#')
- concatenation of lines ending with an escape ('\')
- separation of command lines joined on the line using semicolon (';')
- expansion of textual substitutions introduced with an expansion escape ('$')

- incorporation in to a single command line of every line that is enclosed by brackets (and), { and }, < and > , and double quoted (") strings.

The characters '#', '\' and '$' are not used as part of FTL expression syntax.

Some examples (including the new line prompt '> ') might include:

```
> echo Hello World

> echo A Long \
> Hello \
> World

> echo Hello World; echo Hello Again

> set lines "more than
> one line
> with newlines in it"

> set program {
>    echo Hello World
> }

> echo $lines; echo ${lines}
```

The macro expansion is similar to that traditionally found in POSIX shell programs, and is describe in a section below "Macro Expansion".

Command lines usually have a structure (also familiar to those who have used POSIX shells) in which an initial word is taken as the name of a command and further text provides further detail about what is required often with no syntax more complex than further words, numbers and quoted strings. The way that these lines are interpreted in the light of the current values of variables in the FTL language is explained just below "Standard Command Line Parsing".

In a fashion, the interpretation of these command lines amounts to a very simple programming language. That language, however, is not FTL itself.

One consequence of this is that the way a variable is used will be different depending on whether it is used as part of the command line syntax or as part of FTL expression syntax.

Standard Command Line Parsing

The run-time library consists of a number of objects provided by default in the root environment. They fall in to the following categories:

Function	Built-in closure taking a fixed number of FTL arguments
Command	Built-in program that parses its own command line after parameter substitution
Variable	FTL value initialized by the interpreter
Environment	FTL directory value providing values for specific names

These are used in two separate environments:

1. As the command on a command line - one of the initial elements on an interpreted command line
    - When a *function* name is used as the first element in the line the rest of the command line must provide its arguments, separated by spaces. Enough arguments to bind to all unbound variables must be provided (and no more). The function is then executed and the result, if not `NULL`, is printed out.
    - When a *command* name is used as the first element in the line the rest of the command line is `$`-expanded and given to the command to parse. A warning is produced if it completes before using all the characters on the line and the result, if not `NULL`, is otherwise printed out.
    - When an *environment* name is used as the first element in the line that environment is pushed on to the environment stack and the rest of the line is reparsed as a command in that environment (as above).
    - A *variable* name, which does not refer to any of the above types of value can not be used.
2. As an FTL value
    - The value is returned as part of an FTL expression.

These are some examples of command lines:

```
> command anything
> directory subdirectory-in-directory command-in-subdirectory any syntax at
all
> function FTL-expression-1 FTL-expression-2
> directory FTL-expression-in-directory-1 FTL-expression-in-directory-2
```

Because the primary aim of the language is to provide a command interpreter most input files are interpreted as scripts in the script syntax, with FTL expressions used only when required for command definition and extension.

FTL Literals Syntax

```
<code_literal> ::= '{' [ '\' <char> | <char> ]* ] '}'
<string> ::= '"' ['\' <escape> | <char> ]* '"'
<integer> ::= [<base>] <digit> (<digit> | '_')*
<base> ::= '0x' | '0X' | '0o' | '0O' | '0b' | '0B'
<identifier> ::= <common_identifier> | <builtin_identifier>
<common_identifier> ::= <alphabetic> (<alphabetic> | '_' | <digit>)*
<builtin_identifier> ::= '_' <digit>+
```

In `<code_literal>`s all the characters between an initial '{' and a matching '}' are used except '\' which is used to ensure the following character is included (e.g. '\\' will include the '\' character). A '}' matches the initial '{' only when it does not occur within bracketed text (i.e. text between '{' and '}', or between an initial double quote (") and its matching closing double quote).

In a `<string>` all characters between an initial double quote (") and its matching closing double quote are included except '\' which is used to introduce one or more characters in an escape sequence.

For <integers> the numeric base 16 is specified by '0x' and '0X'; the base 8 is specified by '0o' and '0O'; and the base 2 is specified by '0b' and '0B'. Integers beginning with a numeric base specification use digits from '0' up to a character representing one less than the base. The characters 'A', 'B', 'C', 'D', 'E' and 'F', or 'a', 'b', 'c', 'd', 'e', and 'f' are used for digits of value 10, 11, 12, 13, 14, and 15 as required by base 16. Following the initial digit the character '_' can be used – it is ignored in the value and is accepted only to aid the comprehension of long numbers (e.g. it might be used every three characters as a thousands separator, or every eight binary digits as a byte separator).

FTL Expression Syntax

```
<expression> ::= (<index> '=')* <invocation>
<invocation> ::= (<substitution> ['!']*)+
<substitution> ::= ['&'] <operation>+
<operation> ::= user_operator_parse(<retrieval> | <reference>)
<reference> ::= '@' <retrieval>
<retrieval> ::= [<closure>]['.' <fieldsindex>]*
<fieldsindex> ::= '(' <expression> ')' |
                  <index> |
                  <vector> |
                  <id_environment>
<closure> ::= (<base> [':' | '::'])* <base>
<base> ::= '(' <expression> ')' |
           <code> |
           <id_environment> |
           <identifier> |
           <vector> |
           <type_literal>
<code> ::= <code_literal>
<expression_list> ::= <expression> (';' <expression>)* [';']
<id_environment> ::= '[' [<binding_list> | <binding_list> ',' <unbound_list>
| <unbound_list>] ']'
<binding_list> ::= [<binding> (',' <binding>)*]
<unbound_list> ::= <index> (',' <index>)*
<binding> ::= <index> '=' <invocation>
<vector> ::= '<' <implied_series> | <series> '>'
<index> ::= <identifier> | <type_literal>
<type_literal> ::= <integer> |
                   <string>
<series> ::= [<series_binding> (',' <seres_binding>)*]
<implied_series> ::= [<series_binding> [',' <series_binding>]] '..'
[<series_binding>]
<series_binding> ::= [<invocation> '='] <invocation>
```

In a <code> value the <code_literal> is interpreted as an <expression_list>.

FTL Expression evaluation

User-Operator Parsing

User-operators are provided as a convenience to the language user. The functionality of the language is completely accessible without them.

User-operator parsing is determined by the set of operations defined using `parse.opset` (described as part of the run-time library below). Each operator definition links the name of an operator (which may be one or more characters and can involve unused punctuation characters or identifiers) with a precedence, an associativity and a monadic or diadic binding. The associativity determines whether the operator is a prefix, infix, or postfix one and which of its arguments may be parsed at the same precedence . The precedence defines how tightly the arguments of the operator are bound to the operator in comparison with other operators. The binding should take either one (for a monadic operator) or two (for a dyadic operator) arguments and will be executed once the operator's arguments have been parsed successfully.

Standard Operator Definitions

A number of user-operators are defined as part of the basic run-time library, but these may be removed, changed or added to. Operator definitions are held in the priority-ordered vector `parse.op`, which initially contains:

Priority	Operator	Function	Assoc
0	_or_	logor	xfy
1	_and_	logand	xfy
2	_not_	invert	fy
3	_bitor_	bitor	xfy
4	_bitxor_	bitxor	xfy
5	_bitand_	bitand	xfy
6	_bitnot_	bitnot	fy
7	==	equal	xfx
	!=	notequal	xfx
	_lt_	less	xfx
	_le_	lesseq	xfx
	_gt_	more	xfx
	_ge_	moreeq	xfx
8	_shl_	shiftl	xfy
	shr	shiftr	xfy
9	-	neg	fy
10	+	add	xfy

	-	sub	xfy
11	*	mul	xfy
	/	div	xfy

Note that the priorities refer to the relative position in the vector parse.op - their absolute value is not relevant.

Default Parsing

After parsing user-operators all evaluation in FTL occurs in an `<invocation>`

- where a final '`!`' has been used. In an invocation a `<code>` value or a `<closure>` is executed; or,
- where the `<invocation>` is marked for automatic execution (e.g. using the `func` command) and the `<invocation>` has no unbound variables.

When a `<code>` value is executed the associated text is interpreted as an `<expression_list>` in the current environment (i.e. so that names within the code body will have the values currently in scope at the point of invocation).

A `<closure>` is an association between a `<code>` value and an environment, which is normally constructed from a stack of component environments. The environment can consist both of names with associated values (bound variables) and names with no current value (unbound variables). Only closures with no unbound variables can be executed. When it is executed the environment at the point of invocation replaced by the environment associated with the closure and the `<code>` value is then executed in that environment. The values in scope are those currently available in the (stacked) closure environments - not the values that were defined in those environments at the time of definition of the closure (see below).

There are two operators provided to create a `<closure>` both involving the specification of arguments that are an environment or `<code>` values. The first is the double-colon operator (`::`), which creates a closure whose environment is precisely what has been specified by the environment(s) supplied. The second is the colon operator (`:`), which creates a closure whose environment is the provided value stacked on to the current environment (when the colon operator is executed).

Although it is not currently precisely the case, it is intended that all the symbols (e.g. eventually including numeric and string operators and constants) in a `<code>` value are interpreted relative to the environment in which they execute.

FTL Variable Scope

At any given time the currently accessible scope is defined as an environment that is normally itself constructed as a stack of other environments or simple directories. Within a directory some name bindings are constant - which means that the value associated with the name can not be changed, Also, some stack environments may be "locked" - which means that they can not be extended with new values, and names can not be removed from them.

A directory can contain only one value for a name but, because the same name can be used in different directories in a stacked environment it is possible that there will be more than one

name-value pair in the environment. When a name is used in an environment it refers to the value the name given in the most recently pushed directory, independently of whether it is constant or not (writing to a constant value will fail).

The only visible value for 'a' in this environment

```
[a=X]::[a=Y]::[a=Z]
```

will be Z.

If a value is being provided for a new name, without specifying a parent directory, and no name is found in the current environment then the directory used to store the new name in is the one most recently pushed, independently of whether it is locked or not (so creating a new name-value pair when the last pushed directory is locked will fail).

The empty directory name as a parent specifies the entire environment. This notation can be used to create variable names using the FTL indexing notation (see below). Thus "`.name`" will refer to the same variable that `name` does.

FTL Indexing

The dot ('`.`') provides a couple of syntactical short-cuts. Normally

```
X.Y
```

is equivalent to

```
X::{Y}!
```

That is, "Y" is interpreted in the context of the environment "X". Thus if X is an environment containing strings `X."string"` will return the value associated with the string "`string`" and if it contains integers `X.integer` refers to the value associated with the integer `integer`. For convenience FTL also allows the syntax `X.identifier` as an equivalent to `X."identifier"`. Not all strings are identifiers: identifiers are those which begin with an alphabetic character and then continue with characters that are either alphanumeric or the character '`_`'.

Similarly if Y is itself a literal environment each of the values in it will be interpreted as names in X and will be replaced by their values in X. This applies recursively to any values in Y that are environments. Currently the convenience of using string values without a string denotation does not also apply to literal environments.

The syntax

```
X.(Y)
```

can be used when a literal value is not available. The value of expression Y is used as an object to be interpreted in the context of X. If this value is an integer, string or environment it is treated in the way indicated above.

Examples:

```
> set vec <2,6,9>
> set ids [a=6,b=2,h=9]
> eval vec.0
```

```
2
> eval ids.h
9
> eval ids."h"
9
> eval vec.(ids.b)
9
> eval vec.<2,1,2>
<9, 6, 9>
> eval vec.[f=2,g=1,h=2]
[f=9, g=6, h=9]
> eval ids.<"b","a">
<2, 6>
> eval ids.[q="h", l="a"]
[q=9, l=6]
> eval vec.<4>
<>
> eval vec.<4,0,1>
<1=2, 6>
> eval vec.(ids)
[a=, b=9, h=]
> eval ids.[x="i", y="b"]
[x=, y=2]
> inenv ids.[x="i", y="b"] "x"
FALSE
> inenv ids.[x="i", y="b"] "y"
TRUE
```

FTL Local Variables

A directory can be used as a set of local variable names using the `enter` built in function (see below); or a closure with local variables as its arguments can be used. For example a function to check whether a string is empty that uses a local variable "`enter`" could be written like this:

```
set is_empty [string]: {
    [answer = FALSE]:{
        (len string!) == 0 {answer = TRUE}!;
        answer}!
}
```

or using "`enter`":

```
set is_empty [string]: {
    enter [answer = FALSE]!;
    (len string!) == 0 {answer = TRUE}!;
    answer
```

```
}
```

The "`leave`" or "`leaving`" functions return to the previous environment as it was before a matching `enter`. The difference between them is that leaving returns the environment that was built up since the matching `enter`.

```
set make_imaginary [re,im]: {
    enter [numbertype = "complex"]!;
    .real = re;
    .imaginary = im;
    leaving!
}
> make_imaginary 3 4
[numbertype="complex",real=3,imaginary=4]
```

FTL References

Left hand values (references) can be established with the '@' operator placed before a <name> or retrieval consisting of a number of names concatenated using '`.`'. The reference is closure that takes a single argument whose value will be assigned to the name when it is executed. When the left hand value of something specified with more than one concatenated name is taken the object specified by the names up to the last must exist, and will constitute the environment in which the final name is assigned when the reference is executed. The following are equivalent:

```
@a 20!
```

and

```
a = 20
```

FTL Values

Each value associated with a name belongs to one of the value types supported. These types, and the values they support include the following:

Type	Values
nul	This type has only a single value, `NULL`
type	This type contains different values corresponding to each of the supported types.
int	This type is used for integers. The integers supported are those that can be represented in 64 bits. Denotations are available for specifying both positive and negative integer values however denotations that appear more than `9223372036854775807` (`0x7fffffffffffffff`) can be expected to be indistinguishable from values for negative numbers.
bool	This type contains the values `TRUE` and `FALSE`.

string	This type is used for a sequence of 8-bit octets (bytes). Memory-size permitting, the sequence can contain an implementation-defined number of bytes which must be at least 2^31. The bytes in the sequence can contain any 8-bit value (including zero). Denotations are available that allow specific hexadecimal byte values to be inserted, byte values taken from the default character set; and, multi-byte sequences corresponding both to specific characters from a wide (normally Unicode) character set and from the input character set. In general the run-time library provides support for access to strings treated as octet containers separately from those treated as characters containers. Each character is encoded as a fixed sized, possibly multi-byte, sequence of octets.
code	This type is used for program text. Normally this text can not be "executed" until an environment in which its values can be interpreted is supplied.
dir	This type is used to contain a mapping between "names" and values. The names are themselves values and these types must come from types that support a comparison operator. Currently the types are limited to integers and strings. Currently separate denotations are available for string indexed directories, integer indexed directories and integer indexed integer sequences. Currently the storage requirement for an integer indexed directory is proportional to the maximum index used, and only positive indeces are supported. In addition to "names" with associated values directories can also contain an ordered set of unbound names with no associated value.
closure	This type incorporates both a code value and a dir value that specifies the environment in which code should be interpreted. Values can be bound to unbound names held in their dir values. These values can not be "executed" unless there are no unbound names in their dir value. When they are executed their code value is interpreted in the environment specified by their dir value. Closures can be marked for automatic execution. If they are not marked for automatic execution, execution requires an explicit “!”. Otherwise execution will occur automatically when the closure has no unbound names.
cmd	This type supports different kind of executable value from a closure. Each value incorporates an executable program which will parse a single string argument. A number of commands are built-in in the run time library.
function	This type supports built-in closure values that use a fixed number of arguments taken from a specific range of names ("_1", "_2" ... etc.). The run-time library provides a number of these values.
stream	This type supports values that can either supply or consume a series of octets (bytes).
ipaddr	This type is used to contain an Internet IP address. Denotations are available both for specifying the address as a "dotted quad" or an Internet host name. Currently only addresses for IPv4 are supported.
macaddr	This type is used to contain an IEEE 802.3 MAC address. A denotation is available for specifying the address as a colon-separated sextet. Currently only 48-bit addresses are supported.

Boolean Values

In this version of FTL the values of TRUE and FALSE are:

```
TRUE = [val]::{val!}
FALSE = [val]::{FALSE}
```

That is, TRUE is a closure that will execute its argument and FALSE is a closure that will discard its argument and return FALSE instead. Thus if a value that is either TRUE or FALSE is given code as an argument:

```
<true_or_false> { <code> }!
```

the result will be either the result of the code's execution or the value FALSE. Thus the code is conditionally executed.
A compound boolean expression can be built up in which each item in the chain is executed only if previous ones have evaluated to TRUE:

```
<true_or_false> {<boolean_expression>}! {<boolean_expression>}! ...
{<boolean_expression>}!
```

Thus this construct can be used as a conditionally executed "and".

Automatic Execution of Closures

The most explicit form of FTL does not use automatic execution of closures: a “!” is always required to indicate when a closure is to be executed, even if it has no unbound variables. For example the built in function add has two unbound variables:

```
> eval add
[_1, _2]::<func:0x7fe091a2856b,2>
> eval add 1
[_2]::<func:0x7fe091a2856b,2>
> eval add 1 2
[]::<func:0x7fe091a2856b,2>
> eval add 1 2 !
3
```

While this makes the meaning of code very clear the “!” is nearly redundant – in the sense that, once an argument has been applied that assigns the last unbound variable, it is almost always then executed. For this reason, for convenience, a closure can be marked for automatic execution, whereby such closures are executed automatically if they occur with no bound variables.

None of the built-in functions have closure values values marked for automatic execution.

The func command is provided to mark a normal closure as one for automatic execution. One closure created from another by substituting a value for an unbound variable will inherit the marking. For example

```
> func fnadd[a,b]:{add a b!}
> eval fnadd
[a, b]:{add a b!}
> eval fnadd 1
[b]:{add a b!}
> eval fnadd 1 2
3
```

Because a closure with no unbound variables can be useful as a value, there is an operator ‘&’ that can be used to return a closure which is not marked for automatic execution. For example:

```
> eval &fnadd 1
[b]:{add a b!}
> eval &fnadd 1 2
[]:{add a b!}
> eval &fnadd 1 2 !
3
```

Macro Expansion

The command line handler incorporates a traditional-style macro handler which uses the '$' character to introduce a macro invocation.

What a Macro Invocation Looks Like

Macro invocation uses one of these syntaxes:

- $<decimal_number> - expand the value of FTL expression "`.<decimal_number>`"
- $<identifier> - expand the value of FTL expression "`<identifier>`"
- ${<expression>} - expand the value of FTL expression "`<expression>`"

The expression `.1`, as a <decimal_number>, for example will return the value of symbol `1` in top level of the current environment. This means that, when a series of environments have been stacked (as might occur when integers are used as unbound variables in a series of closures which have invoked each other) `.1` will refer only to the current invocation.

The expression `var`, as an <identifier>, will refer to the current value of that symbol (at any level in the environment).

Note that the first two syntaxes can be re-expressed using the third, since $<decimal_number> is the same as ${.<decimal_number>} and $<identifier> is the same as ${<identifier>}. This can be convenient when the macro is to be expanded in text which might be consumed as part of a <decimal_number> or an <identifier>. For example:

- '12 ${item}s' can be used in place of the otherwise ambiguous '12 $items'
- '${.1}000 bytes' could be used in place of the otherwise ambiguous '$1000 bytes'

When an expression is expanded its value is calculated and then the macro invocation characters are replaced by text. If the value is a string that string forms the text of the replacement. Otherwise the replacement text will be the text with which that value would normally be printed.

When Macro Expansion Takes Place

Macro expansion occurs on two separate occasions:

1. When using the standard command line parsing method: after each line of input is complete (e.g. once a number of input lines using '\' at the end of each one are concatenated, or the lines between matching brackets are concatenated).
2. Before a command is invoked as part of an FTL expression the expansion is applied to its argument. (When commands, which usually consume the text of the rest of the line when used on a command line, are used in this way they take a single string argument.)

In either case macros invocations are sought throughout the text except:

- if the leading '$' is preceded by a backslash (\) character; or,
- if the invocation occurs in a section of the text bracketed by '{' and '}'
  - where the final '}' matches the first '{' in terms of included '{ .. }' sections
  - where the initial '{' is not preceded by a backslash (\) character.

Naturally, the result of a macro expansion depends on the values of symbols in the environment when the expansion takes place.

In the first case, of a completed line of input, the environment is the one left by any previous line. This is true even if the line includes a prefix which denotes a change or restriction of the current environment.

In the second case the environment used is the latest one, which includes the binding of a single unbound variable representing the command's string argument.

These examples use the standard command line parsing method:

```
> set name "Eric"
> echo Your name is $name
Your name is Eric
> echo {Your name is $name}
{Your name is $name}
> set hi[name]:{echo "Hello $name"!}
> hi "sunshine"
Hello sunshine

> echo hi is $hi
hi is [name]:{echo "Hello $name"!}

> set age 50
> eval echo "Hi $name, happy ${age}th"!
Hi Eric, happy 50th
> eval echo {Hi $name, happy ${age}th}!
Hi Eric, happy 50th
> set person [name="Sunny Joe", age=30]
> person eval echo "Hi $name, happy ${age}th"!
Hi Eric, happy 50th
> person eval echo {Hi $name, happy ${age}th}!
Hi Sunny Joe, happy 30th

> set args <"copy", "fileA", "fileB">
> echo Program ${args.0}
Program copy
> args eval echo {${.0}ing $1 to $2}!
copying fileA to fileB
> set log[0,1,2]:{ echo ${.0}ing $1 to $2"! }
> log "copy" "fileA" "fileB"
copying fileA to fileB
```

Run-time library overview

The values available in the run time library include those in the following categories.

- Program control
    - catch <excep> <do> - run <do> executing <excep> <ex> on an exception
    - do <do> <test> - execute <do> while <test> evaluates non-zero
    - every <n> <command> - repeat command every <n> ms
    - exit - abandon all command inputs
    - for <env> <binding> - execute <binding> for all <env> values
    - forall <env> <binding> - execute <binding> <val> <name> for all entries in <env>
    - forwhile <env> <binding> - for <binding> while it is not FALSE
    - forallwhile <env> <binding> - forall <binding> while it is not FALSE
    - if <b> <then-code> <else-code> - execute <then-code> if <n>!=FALSE
    - return <rc> - abandon current command input returning <rc>
    - source <filename> - read from file <filename>
    - sourcetext <strin>|<code> - read characters from string or code
    - throw <value> - signal an exception with <value>, exit outer 'catch'
    - while <test> <do> - while <test> evaluates non-zero execute <do>
- General
    - cmd <function> <help> - create a command from a function
    - cmp <expr> <expr> - returns integer comparing its arguments
    - command <whole line> - execute as a line of source
    - echo <whole line> - prints the line out
    - eval <expr> - evaluate expression
    - func <name> <closure> - set auto exec closure value in environment
    - help - prints command information
    - set <name> <expr> - set value in environment
    - sleep <n> - sleep for <n> milliseconds
- Types
    - basetype - directory of built in type values
    - NULL - nul value
    - type <typename> - return the type with the given name
    - typeof <expr> - returns the type of <expr>
    - typename <expr> - returns the name of the type of <expr>
- Booleans
    - TRUE - TRUE value (an un-FALSE value)
    - FALSE - the FALSE value
    - equal <val> <val> - TRUE if first <val> equal to second
    - invert <val> - TRUE if <val> is FALSE, FALSE otherwise
    - less <val> <val> - TRUE if first <val> less than second
    - lesseq <val> <val> - TRUE if first <val> less than or equal to second
    - logand <val1> <val2> - FALSE if <val1> is FALSE, <val2> otherwise
    - logor <val1> <val2> - TRUE if <val1> is TRUE, <val2> otherwise
    - more <val> <val> - TRUE if first <val> more than second
    - moreeq <val> <val> - TRUE if first <val> more than or equal to second
    - notequal <val> <val> - TRUE if first <val> not equal to second
- Integers
    - add <n1> <n2> - return n1 with n2 added
    - bitor <n1> <n2> - return n1 "or"ed with n2
    - bitxor <n1> <n2> - return n1 exclusive "or"ed with n2
    - bitand <n1> <n2> - return n1 "and"ed with n2
    - bitnot <n> - return the bitwise "not" of n

    - div <n1> <n2> - return n1 divided by n2
    - int <integer expr> - numeric value
    - intseq <first> <inc> <last> - vector of integers incrementing by <inc>
    - int_fmt_hexbits <n> - print decimal if all bits in this mask, else hex
    - mul <n1> <n2> - return n1 multiplied by n2
    - neg <n> - return negated <n>
    - rndseed <n> | <string> - set random seed based on argument
    - rndseq <n> - vector of integers containing 0..<n>-1 in a random order
    - rnd <n> - return random number less than <n>
    - shiftl <n1> <n2> - return n1 left shifted by n2 bits
    - shiftr <n1> <n2> - return n1 right shifted by n2 bits
    - sub <n1> <n2> - return n1 with n2 subtracted
- Environments and directories
    - domain <env> - generate vector of names in <env>
    - dynenv <getall> <count> <set> <get> - dynamic env built from given functions
    - enter <env> - add commands from <env> to current environment
    - envjoin <env1> <env2> - composed directory indexing <env2> with <env1> values
    - inenv <env> <name> - returns 0 unless string <name> is in <env>
    - leave - exit the environment last entered or restricted
    - leaving - exit the environment last entered or restricted & return it
    - len [<env>|<closure>|<string>] - number of elements in object
    - lock <env> - prevent new names being added to <env>
    - new <env> - copy all <env> values
    - range <env> - generate vector of values in <env>
    - restrict <env> - restrict further commands to those in <env>
    - rndseq <n> - vector of integers containing 0..<n>-1 in a random order
    - select <binding> <env> - subset of <env> for which <binding> returns TRUE
    - sort <env> - sorted vector of values in <env>
    - sortby <cmpfn> <env> - sorted vector of values in <env> using <cmpfn>
    - sortdom <env> - sorted vector of names in <env>
    - sortdomby <cmpfn> <env> - sorted vector of names in <env> using <cmpfn>
    - zip <dom> <range> - generate environment with given domain and range taken from <range>
- Closures
    - argname <closure> - return name of 1st argument to be bound or NULL
    - argnames <closure> - return vector of arguments to be bound
    - bind <closure> <arg> - bind argument to unbound closure argument
    - closure <push> <dir> <code> - create closure from code and dir (+ stack if push)
    - code <closure> - return code component of a closure
    - context <closure> - return environment component of a closure
    - deprime <closure> - unmark closure for automatic execution
    - func <name> <closure> - set auto exec closure value in environment
    - prime <closure> - mark closure for automatic execution when all args bound
- Character handling
    - binsplit <le?> <signed?> <n> <str> - make vector of <signed>> <n>-byte ints with <le?> endianness
    - chr <int> - returns string of (multibyte) char with given ordinal
    - chrcode <string> - returns ordinal of the first character of string

    - collate <str1> <str2> - compare collating sequence of chars in strings
    - chop <stride> <str> - make vector of strings each <stride> bytes or less
    - chopn <n> <stride> <str> - make vector of <n> strings each <stride> bytes or less
    - len [<env>|<closure>|<string>] - number of elements in object
    - join <delim> <str> - join vector of octets and strings separated by <delim>s
    - joinchr <delim> <str> - join vector of chars and strings separated by <delim>s
    - octet <int> - returns single byte string containing given octet
    - octetcode <string> - returns ordinal of the first byte of string
    - split <delim> <str> - make vector of strings separated by <delim>s
    - str <expr> - evaluate expression and return string representation
    - strf <fmt> <env> - formatted string from values in <env>
    - strcoll <str1> <str2> - compare collating sequence of chars in strings
    - strlen [<string>] - number of (possibly multibyte) chars in string
    - tolower <str> - lower case version of string
    - toupper <str> - upper case version of string
- Binary data handling
    - binsplit <le?> <signed?> <n> <str> - make vector of <signed>> <n>-byte ints with <le?> endianness
    - mem base_can <mem> [rwgc] <ix> - return start of area ending at <ix> that can do ops
    - mem base_cant <mem> [rwgc] <ix> - return start of area ending at <ix> that can do ops
    - mem bin <base> <string> - create mem with base index and read-only string
    - mem block <base> <len> - create block of mem with len bytes and base index
    - mem dump <+char?> <ln2entryb> <mem> <ix> <len> - dump content of memory
    - mem get <mem> <ix> <len> - force read <len> string at <ix> in memory
    - mem len_can <mem> [rwgc] <ix> - return length of area at <ix> that can do ops
    - mem len_cant <mem> [rwgc] <ix> - return length of area at <ix> that can not do ops
    - mem read <mem> <ix> <len> - read <len> string at <ix> in memory
    - mem rebase <mem> <base> - place <mem> at byte index <base>
    - mem write <mem> <ix> <str> - write binary string at index to memory
- Input and Output
    - io binfile <filename> <rw> - return stream for opened binary file
    - io close <stream> - close stream
    - io connect <protocol> <netaddress> <rw> - return stream for remote connection
    - io err - default error stream
    - io file <filename> <rw> - return stream for opened file
    - io filetostring <filename> [<outfile>] - write file out as a C string
    - io flush <stream> - ensure unbuffered output is written
    - io fmt - updateable directory of formatting functions for io.fprintf
    - io fprintf <stream> <format> <env> - write formatted string to stream
    - io getc <stream> - read the next character from the stream
    - io in - default input stream
    - io inblocked <stream> - TRUE if reading would block
    - io instring <string> <rw> - return stream for reading string
    - io listen <protocol> <netport> <rw> - return stream for local port
    - io out - default output stream
    - io outstring <closure> - apply output stream to closure and return string

    - io pathfile <path> <filename> <rw> - return stream for opened file on path
    - io pathbinfile <path> <filename> <rw> - return stream for opened binary file on path
    - io read <stream> <size> - read up to <size> bytes from stream
    - io ready <stream> - return whether next write may not cause wait
    - io stringify <stream> <expr> - write FTL representation to stream
    - io write <stream> <string> - write string to stream
- Parser interface
    - parse argv - directory of command line arguments for command
    - parse assoc - environment containing operator associativity definitions
    - parse codeid - name of interpreter
    - parse env - return current invocation environment
    - parse errors - total number of errors encountered by parser
    - parse errors_reset <n> - reset total number of errors to <n>
    - parse exec <cmds> <stream> - return value of executing initial <cmds> then stream
    - parse execargv - directory of command line arguments given to interpreter
    - parse expand <inc{}> <env> <val> - expand macros in string or code <val>
    - parse line - number of the line in the character source
    - parse local - return local current invocation directory
    - parse newerror - register the occurrance of a new error
    - parse op - environment containing operation definitions
    - parse opeval <opdefs> <code> - execute code according to operator definitions
    - parse opset <opdefs> <prec> <assoc> <name> <function> - define an operator in opdefs
    - parse readline - read and expand a line from the command input
    - parse rdenv <cmds> <stream> - return env made executing initial <cmds> then stream
    - parse rdmod <module> - return env made executing module file on path
    - parse root - return current root environment
    - parse scan <string> - return parse object from string
    - parse scancode <@string> <parseobj> - parse {code} block from parse object
    - parse scanempty <parseobj> - parse empty line from string from parse object, update string
    - parse scanspace <parseobj> - parse over white space from string from parse object, update string
    - parse scanhex <@int> <parseobj> - parse hex string from parse object, update string
    - parse scanhexw <width> <@int> <parseobj> - parse hex in <width> chars from parse object, update string
    - parse scanid <@string> <parseobj> - parse identifier, update string
    - parse scanint <@int> <parseobj> - parse integer from string from parse object, update string
    - parse scanintval <@int> <parseobj> - parse signed based integer from parse object
    - parse scanitem <delims> <@string> <parseobj> - parse item until delimiter, update string
    - parse scanitemstr <@string> <parseobj> - parse item or string, update string

    - parse scanmatch <dir> <@val> <parseobj> - parse prefix in dir from parse object giving matching value
    - parse scanned <parseobj> - return text remaining in parse object
    - scanopterm <opdefs> <scanfn> <@val> <parseobj> - parse term using <opdefs> & base <scanfn>
    - parse scanstr <@string> <parseobj> - parse item until delimiter, update string
    - scantomatch <dir> <@val> <parseobj> - parse up to delimiter named in dir in <parseobj> giving matching value
    - parse scanvalue <@string> <parseobj> - parse a basic value in <parseobj>
    - parse scanwhite <parseobj> - parse white spce from string from parse object, update string
    - parse source - name of the source of chars at start of the last line
    - parse stack - return local current invocation directory stack
- Operating System interface
    - sys env - system environment variable environment
    - sys fs casematch - whether case must match in file system file names value
    - sys fs absname <file> - TRUE iff file has an absolute path name
    - sys fs home - the user's home directory name
    - sys fs nowhere - name of file available as empty source or sink
    - sys fs rcfile - name of the file, in the home directory, holding initial commands
    - sys fs sep - string separating path elements in a file name
    - sys fs thisdir - name of directory representing the current directory
    - sys localtime <time> - broken down local time
    - sys localtimef <format> <time> - formatted local time
    - sys osfamily - name of operating system type
    - sys run <line> - execute system <line>
    - sys runrc <command> - execute system command & return result code
    - sys shell path <path> <file> - return name of file on path
    - sys shell pathsep - character value separating directory elements in a path environment
    - sys shell self - file system name of interpreter binary
    - sys ticks - current elapsed time measure in ticks
    - sys ticks_hz - number of ticks per second
    - sys ticks_start - ticks value when interpreter was started
    - sys time - system calendar time in seconds
    - sys uid <user> - return the UID of the named user
    - sys utctime <time> - broken down UTC time
    - sys utctimef <format> <time> - formatted UTC time
- Windows-only interface
    - reg value HKEY_CLASSES_ROOT <key> - open key values in HKEY_CLASSES_ROOT
    - reg value HKEY_USERS <key> - open key values in HKEY_USERS
    - reg value HKEY_CURRENT_USER <key> - open key values in HKEY_CURRENT_USER
    - reg value HKEY_LOCAL_MACHINE <key> - open key values in HKEY_LOCAL_MACHINE
    - reg value HKEY_CURRENT_CONFIG <key> - open key values in HKEY_CURRENT_CONFIG
    - reg allow_edit <boolean> - enable/disable write to the registry from new keys

- reg key - directory of key value types

Run-time Library

In the following each library entity is described using a proforma including a specification of the argument syntax. This uses "<label:syntax>" to refer to something parsed according to syntax indicated in the following table that is then referred to in a description using the name "label".

any	FTL expression returning any type of value
int	FTL expression returning an integer value
bool	FTL expression returning either `TRUE` or `FALSE`
code	FTL expression returning a code value
dir	FTL expression rerturning a directory value
closure	FTL expression rerturning a closure value
clodir	<closure> | <dir>
clocode	<closure> | <code>
intordir	<int> | <dir>
string	FTL expression rerturning a string value
stringnl	FTL expression rerturning a string, NULL or integer
stringorcode	FTL expression rerturning either a string or a code value
token	Any sequence of characters with no white space
tokstring	Either a <token> or a literal string
restofline	The sequence of characters starting at the first non-white character and ending at the end of the line. Note: when supplied to a command inside an FTL expression this is supplied as a string value or a code value. When a string is used it is subject to macro expansion before being provided to the command.

The names in the default run-time library are presented in alphabetic order below.

add <int> <int>

Name	`add`
Kind	Function
Arg syntax	<n1:int> <n2:int>
Description	Takes two integers and adds them together.
Returns	An integer with the value n1+n2. See also: `sub`, `mul`, `div`, `shiftl`, `shiftr, neg`

Example	`> add 0x100 12`
	`268`
	`> sys localtimef "%T" (add (sys.time!) 600!)`

```
"17:30:14"
```

argname <closure>

Name `argname`

Kind Function

Arg syntax <closure>

Description Return only the unbound variable part of a closure (e.g. a function) that would be bound to an argument.
There are three main portions of a closure: its bound environment, its unbound environment and its code. This function returns the last created unbound name. If there are no unbound variable NULL is returned.
See also: `bind`, `code`, `context`, `closure`, `argnames`

Returns A string containing the name of the variable next to be bound to an argument.

Example
```
> argname [b=1,a]::{a+b}
"a"
> argname set
"_1"
> set needs_argument[fn]:{NULL != (argname fn!)}
> needs_argument needs_argument
TRUE
> needs_argument []:{echo "no"!}
FALSE
```

argnames <closure>

Name `argnames`

Kind Function

Arg syntax <closure>

Description Returns a vector containing only the names of the unbound variables in a closure (e.g. a function).
There are three main portions of a closure: its bound environment, its unbound environment and its code. This function returns the unbound names. The order of the names is the same as the order in which they will be taken by succeeding bindings (i.e. the first element in the sequence is the name of the variable that will be bound first - this is the value that '`argname`' would return). If there are no unbound variable an empty vector is returned.
See also: `bind`, `code`, `context`, `closure`, `argname`

Returns A string containing the name of the variable next to be bound to an argument.

Example

```
> argnames [b=1,a]::{a+b}
<"a">
> argnames add
<"_1", "_2">
> set argument_count[fn]:{len (argnames fn!)!}
> argument_count add
2
```

basetype

Name `basetype`

Kind Directory

Arg syntax Not a function

Description Contains the values of types built-in to the parser.
When printed, types contain the names held by this directory.

Returns Not a function.
See also:`type`, `typename`, `typeof`

Example

```
> typename “some text”
“string”
> typeof "some text"
$basetype.string
> eval basetype.string
$basetype.string

> help all basetype
all basetype <subcommand> - commands:
    type - type value
    nul - type value
    string - type value
    code - type value
    closure - type value
    int - type value
    dir - type value
    cmd - type value
    fn - type value
    stream - type value
    ip - type value
    mac - type value
    coroutine - type value
    mem - type value
```

bind <closure> <any>

Name `bind`

Kind Function

Arg syntax <fn:closure> <arg:any>

Description Takes a closure with at least one unbound variable and binds the given value to the next to be bound.

Returns A closure that has a previously unbound variable bound to the given value. Note that this does not cause the resulting closure to be invoked even if there are no remaining unbound variables in the closure.
This is an explicit function to represent what normally occurs when using the syntax: <fn> <arg>. It is equivalent to a function defined:

```
set bind[fn, arg]:{fn arg}
```

See also: `closure`, `code`, `context`, `argname`, `argnames`

Example

```
> set dir bind [a] 33!
> eval dir
[a=33]

> set inc bind add 1!
> inc 3
4
```

binsplit <bool> <bool> <int> <string>

Name `binsplit`

Kind Function

Arg syntax <le:bool> <signed:bool> <bytes:int> <data:string>

Description Creates a read-only vector of ‘bytes’ length integers taken from the provided data. Each ‘bytes’-length integer is created from consecutive bytes in the data treated as a little-endian value if le is TRUE and a big-endian value if it is FALSE. Similarly each value is taken as a signed value if ‘signed’ is TRUE and unsigned otherwise.
The result is constructed from whole sequences of ‘bytes’ bytes starting from the first byte. The remainder of the data (that is less than ‘bytes’ in length) is not represented in the resulting vector.

This function creates an alternative representation of the data ‘data’ and does not duplicate the memory required to hold it.

Valid values for ‘bytes’ are only 1, 2, or 4.

Returns Bitwise "and" of `b1` and `b2`.
See also: `chop`

Example

```
> set bin "\xff\xff\xff\xff\x80\x00\x00\x01\x01\x00\x00\xff"
> int_fmt_hexbits 255
-1
> binsplit TRUE TRUE 4 bin
<-1, 0x1000080, 0xfffffffff000001>
> binsplit TRUE FALSE 4 bin
<0xffffffff, 0x1000080, 0xff000001>
> binsplit FALSE TRUE 4 bin
<-1, 0xffffffff80000001, 0x10000ff>
> binsplit FALSE FALSE 4 bin
<0xffffffff, 0x80000001, 0x10000ff>
> binsplit TRUE TRUE 2 bin
<-1, -1, 128, 0x100, 1, 0xffffffffffffff00>
> binsplit TRUE FALSE 2 bin
<0xffff, 0xffff, 128, 0x100, 1, 0xff00>
> binsplit FALSE TRUE 2 bin
<-1, -1, 0xffffffffffff8000, 1, 0x100, 255>
> binsplit FALSE FALSE 2 bin
<0xffff, 0xffff, 0x8000, 1, 0x100, 255>
> binsplit TRUE TRUE 1 bin
<-1, -1, -1, -1, -128, 0, 0, 1, 1, 0, 0, -1>
> binsplit TRUE FALSE 1 bin
<255, 255, 255, 255, 128, 0, 0, 1, 1, 0, 0, 255>
```

bitand <int> <int>

Name `bitand`

Kind Function

Arg syntax <b1:int> <b2:int>

Description Takes two sets of 64 bits (held in the two integers represented in two's complement arithmetic) and returns the integer representing the "and" of corresponding bits.

Returns Bitwise "and" of `b1` and `b2`.
See also: `or`, `bitor`, `bitxor`, `bitnot`

Example

```
> bitand 1 3
1
> bitand 0 0xffff
0
> strf "0x%09x" <bitand 0x000fff000 0x123456789!>
"0x000456000"
```

bitnot <int>

Name `bitnot`

Kind Function

Arg syntax <b:int>

Description Takes a set of 64 bits (held by the integer represented in two's complement arithmetic) and returns the integer representing the "not" of each bit.

Returns Bitwise inversion of `b`.
See also: `and`, `bitor`, `bitxor`, `bitand`

Example
```
> bitnot 0
-1
> bitnot 0xffff
-65536
> strf "0x%09x" <bitnot 0x000fff000!>
"0xfffffffff000fff"
```

bitor <int> <int>

Name `bitor`

Kind Function

Arg syntax <b1:int> <b2:int>

Description Takes two sets of 64 bits (held in the two integers represented in two's complement arithmetic) and returns the integer representing the "or" of corresponding bits.

Returns Bitwise "or" of `b1` and `b2`.
See also: `and`, `bitand`, `bitxor`, `bitnot`

Example
```
> bitor 1 2
3
> bitor 0xffff0000 0xffff
4294967295
> strf "0x%09x" <bitor 0x000fff000 0x123456789!>
"0x123fff789"
```

bitxor <int> <int>

Name `bitxor`

Kind Function

Arg syntax <b1:int> <b2:int>

Description Takes two sets of 64 bits (held in the two integers represented in two's complement arithmetic) and returns the integer representing the "exclusive or" of corresponding bits.

Returns Bitwise "exclusive or" of `b1` and `b2`.
See also: `and`, `bitand`, `bitor`, `bitnot`

Example

```
> bitxor 1 3
2
> bitxor 0xffff 0xff00
255
>  strf "0x%09x" <bitxor 0x000fff000 0x555555555!>
"0x555aaa555"
```

chop <intordir> <string>

Name chop

Kind Function

Arg syntax <stride:intordir> <str:string>

Description Returns a vector of successive substrings from <str> as determined by stride. When stride is an integer each substring is |<stride>| octets long except the last which may have fewer (being all that is left). The substrings are taken sequentially from str with postive values of <stride> being taken from the front of he string and negative ones being taken from the back.

When stride is a directory each element in the vector will have the same shape as the directory (and its subdirectories) except that the values are replaced by successive substrings from <str>.

Returns A vector of successive substrings or directories of substrings taken from the string.
See also: chopn, zip, join

Example

```
> chop 3 "abcdefghijk"
<"abc", "def", "ghi", "jk">
> chop -3 "abcdefghijk"
<"ijk", "fgh", "cde", "ab">
> set rev[x]:{x.<(len x!)-1..0>}
> join "," (rev (chop -3 (strf "%d" <10000000>!)!)!)
"10,000,000"
> chop [word=-4] "thisand that"
<[word="that"], [word="and "], [word="this"]>
> chop [first=<1,1>,body=6] "* thing1**thing2+*thing3"
<[first=<"*", " ">, body="thing1"], [first=<"*", "*">,
body="thing2"], [first=<"+", "*">, body="thing3"]>
> join NULL (chop <-5,-4,-2,-4,-2,-4,-5> "able was I ere I saw
elba "!)
"elba saw I ere I was able "
```

chopn <int> <intordir> <string>

Name chopn

Kind Function

Arg syntax <items:int> <stride:intordir> <str:string>

Description Returns a vector of two items. The first is similar to that returned by `chop` with up to <items> successive substrings from <str> as determined by stride. When stride is an integer each substring is |<stride>| octets long except the last which may have fewer (being all that is left). The substrings are taken sequentially from str with postive values of <stride> being taken from the front of he string and negative ones being taken from the back.

When stride is a directory each element in the vector will have the same shape as the directory (and its subdirectories) except that the values are replaced by successive substrings from <str>.

The second item in the vector returned is a string containing any remaining characters not removed from the string.

Returns A vector of two items. The first a vector of successive substrings or directory of substrings taken from the string; and the second any remaining characters in the string.
See also: `zip, chop, join`

Example

```
> set first[n,str]:{ (chopn 1 n str!).0 }
> set last[n,str]:{ (chopn 1 (-n) str!).0 }
> set allbutfirst[n,str]:{ (chopn 1 n str!).1 }
> set allbutlast[n,str]:{ (chopn 1 (-n) str!}.1 }
> first 3 "abcdefghij"
"abc"
> last 3 "abcdefghij"
"hij"
> allbutfirst 3 "abcdefghij"
"defghij"
> allbutlast 3 "abcdefghij"
"abcdefg"
> chopn 1 [op=1,header=7] "3headerhpacketcontent"
<[op="3", header="headerh"], "packetcontent">
```

chr <int>

Name `chr`

Kind Function

Arg syntax <widechar:int>

Description Returns a string containing a single (possibly-multibyte) character with the given position in the current locale's character set.

Returns A string of octets representing the character with the given ordinal number.
See also: `octet`, `chrcode`

Example

```
> echo ${chr 163!}
£
> chr 163
"\xc2\xa3"
> eval chr 65!
"A"
```

chrcode <char>

Name `chrcode`

Kind Function

Arg syntax <char:string>

Description Returns the ordinal position of the first (possibly-multibyte) character in its argument in the current locale's character set.

Returns An integer containing the ordinal number.
See also: `chr`, `octetcode`

Example

```
> eval "£"
"\xc2\xa3"
> chrcode "\xc2\xa3"
163
> eval chrcode "€"!
8364
```

closure <bool> <dir> <code>

Name `closure`

Kind Function

Arg syntax <bool> <dir> <code>

Description Create a closure from the environment <dir> and code <code>. If <bool> is TRUE the new closure includes the current environment just as
<dir>:<code>
would. Otherwise it does not include the current envionment (i.e. the environment is precisely <dir>) just as
<dir>::<code>
would.
Any unbound variables in <dir> are ignored.
See also: `bind`, `code`, `context`, `argname`, `argnames`

Returns A closure with no unbound variables created from <dir> and <code>.

Example

```
> closure FALSE [inc=3] {<0, inc .. 10>}
[inc=3]::{<0, inc .. 10>}
> set key_chooser[vals]:{(closure FALSE vals {key}!)!}
> key_chooser ["lock"=8,"key"=-9,"bolt"=2]
-9
> key_chooser ["lock"=8,"bolt"=2]
ftl $*console*@26@0: in
ftl $*console*@26: in
ftl $*console*:28: undefined symbol 'key'
ftl $*console*:28: error in closure code body
> set init_42 closure TRUE [first=42] {[value=first]}!
```

cmd <function> <help>

Name `cmd`

Kind Function

Arg syntax <function:closure> <help:string>

Description Create a command from a function that parses the command line as a string. The object created is suitable for entry into a directory of commands and will incorporate a help string as specified. The function provided must be a closure that has precisely one unbound variable which will be bound to a string when it is executed.
This can be used to support command lines which limit the side-effects that arguments can have (since using a function as a command always allow an arbitrary expression as an argument). Alternatively it can be used to support command line syntax which could not be provided simply by parsing a fixed number of expression - such as when names from a given environment are required or when a variable number of arguments are needed.
See also: `parse.scan*` functions

Returns A closure embedding the help string and a new command which evaluates the given function.

Example

```
> set onetwo_fn[s]:{ (split " " s!).<0,1> }
> set onetwo cmd onetwo_fn "<first> <second> .. return 1st and
2nd items"!
> help
...
onetwo <first> <second> .. return 1st and 2nd items
...
> onetwo beaky mick titch
<"beaky", "mick">
> eval onetwo "-o file -- opts"!
<"-o", "file">
```

cmp <any> <any>

Name `cmp`

Kind Function

Arg syntax <arg1:any> <arg2:any>

Description Compares two values returning an integer as follows:

- values incomparable: a non-zero value
- arg1 is arg2: zero
- value of arg1 is equivalent to the value of arg2: zero
- value of arg1 is less than the value of arg2: a negative integer
- value of arg1 is greater than the value of arg2: a strictly positive value

This function does not compare the content of directories or closures. Equality in this case is based on identity (that is a directory or closure is equal only to itself, not another of the same type with identical value).
See also: `equal`, `notequal`, `more`, `less`, `moreeq`, `lesseq`, `collate`

Returns An integer value comparing its arguments.

Example

```
> cmp 9 1
1
> cmp "Astring" "Bstring"
-1
> cmp "string\0_B" "string\0_A"
1
> cmp <1> <1>
-1
> set dir <1>
> cmp dir dir
0
> cmp NULL ({}!)
0
```

code <closure>

Name `code`

Kind Function

Arg syntax <closure>

Description Return only the 'code' portion of a closure (e.g. a function).
There are three main portions of a closure: its bound environment, its unbound environment and its code. This function returns the latter.
See also: `bind`, `closure`, `context`, `argname`, `argnames`

Returns A code value.

Example

```
> code [b=1,a]:{a+b}
{a+b}
> code set
<cmd:0x8063679,1>
> code code
<func:0x805ac4e,1>
```

collate <string1> <string2>

Name `collate`

Kind Function

Arg syntax <chars1:string> <chars2:string>

Description Returns comparison of the lexicographic positions of the two strings derived from the collating sequence of the characters contained in the two strings. The characters may be encoded in more than one octet.
Note that this is a different function to cmp which will compare the octets that constitute each character (which will normally result in a comparison of the ordering of the character's positions in their character set instead of their collating position - which is not always equivalent).
The collating sequence used is provided by the locale in which the interpreter runs.
See also: `cmp`

Returns An integer comparing the lexicographic collating sequences of the two strings.

Example
```
> collate "same" "same"
0
> collate "early" "late"
-1
> sortby collate <"fred", "£", "FORMER", "104">
<3, 2, 0, 1>
```

command <whole line>

Name `command`

Kind Command

Arg syntax <restofline>

Description Execute the text on the rest of the line as if it was a source command.

Normally execution continues until the line is completely read or until a `return` or `exit` command are executed.

As with other commands, when used as a function multiple lines of source can be provided either as a string or as a code value and the text provided is subject to macro expansion in the current environment. This means that, unlike `sourcetext`, macros in the text are expanded before the commands are executed.

See also: `parse.exec`, `source`, `sourcetext`, `return`, `exit`

Returns Any value provided by a `return` command.

Example

```
> set doit "echo"
> command $doit some arguments for $doit
some arguments for echo
> # some arguments for echo
> command return 2+2
4
> #4
>
> set PROFORMA[cond,then]:{cond {command then!}!}
> set WHEN[cond,then]:{cond {sourcetext then!}!}
> set scope 1
> WHEN TRUE {
>     set scope 2
>     echo scope is $scope
> }
scope is 2
> eval scope
2
> set scope 1
> PROFORMA TRUE {
>     set scope 2
>     echo scope is $scope
> }
scope is 1
```

context <closure>

Name `context`

Kind Function

Arg syntax <closure>

Description Return only the 'code' portion of a closure (e.g. a function).
There are three main portions of a closure: its bound environment, its unbound environment and its code. This function returns the latter.
See also: `bind`, `code`, `closure`, `argname`, `argnames`

Returns A directory representing the environment in which <closure> will run.

Example

```
> context [b=1,a]::{a+b}
[b=1, a]
> context set
[_help="<name> <expr> - set value in environment", _1]
> eval (context [b=1,a]:{a+b}!).b
1
> eval (context [b=1,a]:{a+b}!).a
ftl $*console*:12: index symbol undefined in parent '"a"'
ftl $*console*:12: failed to evaluate expression
```

div <int> <int>

Name `div`

Kind Function

Arg syntax <n1:int> <n2:int>

Description Takes two integers and divides the first by the second returning the integer part of the result.

Returns An integer with the value n1/n2.
See also: `add`, `sub`, `mul`, `shiftl`, `shiftr, neg`

Example

```
> div 14 6
2
> div -14 6
-2
> div 14 (-6)
-2
> div -14 (-6)
2
```

do <code> <code>

Name `do`

Kind Function

Arg syntax <do:code> <test:code>

Description execute <do> while <test> evaluates to TRUE

Returns Returns the value associated with <code> on its last execution

Example

```
set nosix_run[]: {
    count = 0;
    do {n = rnd 6!;
        count = 1+(count);
        count
    } { 0 != (n) }!
}
> nosix_run
5
> nosix_run
10
> nosix_run
1
```

domain <dir>

Name `domain`

Kind Function

Arg syntax <env:dir>

Description Provides a means to find just the in-scope names that are bound in env. Names that are hidden because of scoping rules will not appear nor will unbound names.
The order of elements in the vector is undefined, but is the same ordering that is used in the `range` function.
See also: `range`

Returns a vector of names used in `env,` in any order

Example

```
> domain [a=3, b="asdf", "c"=[], d]
<"a", "b", "c">
> join ", " (domain [a=3, b="asdf", "c"=[], d]!)
"a, b, c"
> set rndelement[env]: { env.((domain env!).(rnd (len
env!)!)) }
> rndelement [a=3, b="asdf", "c"=[], d]
"asdf"
> rndelement [a=3, b="asdf", "c"=[], d]
3
> rndelement [a=3, b="asdf", "c"=[], d]
[]
```

equal <val> <val>

Name `equal`

Kind Function

Arg syntax <val1:any> <val2:any>

Description Compares its first argument with the second and returns `TRUE` iff they are equal and `FALSE` otherwise. The comparison is made as described in the `cmp` function.
This function is identical to:

```
[val1, val2]: { 0 == (cmp val1 val2!) }
```

See also: `cmp`, `notequal`, `more`, `less`, `moreeq`, `lesseq`
Normally this function is associated with the == operator.

Returns TRUE or FALSE

Example

```
{   if (equal var NULL!) {
        echo "nothing there"!
    }{  do_something_with var!
    }!
}
> eval (3 == 3) {echo "hi"!}!
hi
FALSE
```

echo <whole line>

Name `echo`

Kind Command

Arg syntax <restofline>

Description Writes the characters following it starting at the first non-blank character to the current output stream followed by a newline.
As with other commands the line is subject to $-expansion first.

Returns NULL

Example

```
> echo   === STARTING ===
=== STARTING ===
> set option "on"
> echo Option is currently $option
Option is currently on
> set follow[a]: { [next = add a 1!]:{ echo "Next $next"! }! }
> follow 4
Next 5
> set copy[1,2]:{ sys.run "echo cp '$1' '$2'"! }
> copy "fileA" "fileB"
cp fileA fileB
```

enter <dir>

Name `enter`

Kind Function

Arg syntax <env:dir>

Description This command pushes its argument directory `env` onto the current environment stack thus bringing all the names defined in the directory into scope.
Since the commands `parse.local` and `help` are active only on the top directory in the current environment the command has a direct effect on them.
Note that (unless left explicitly using `leave` or `leaving`) the environment entered during the execution of a block goes out of scope at the end of the block, together with any pushed directories.
See also: `restrict`, `leave`, `leaving`

Returns NULL

Example

```
> enter [h=help, e=echo, ans=42]
> help
ans - int value
e <whole line> - prints the line out
h - prints command information
> eval h
["_help"="- prints command information"]::<func:0x8053dfc,0>
> eval help
["_help"="- prints command information"]::<func:0x8053dfc,0>
```

```
>
> set f [rec]:{ x="original"; enter rec!; echo x!; }
> f [x="new"]
new
> f [y="new"]
original
>
> eval ans
42
> set f1[]:{ans=80;ans}
> f1
80
> eval ans
80
> set f1[]:{enter [ans=9]!;ans}
> f1
9
> eval ans
80
>
> set item_bill[purchase]: {
    enter purchase!;
        total = items * unit_cost;
    leave!;
    purchase.total
};
> set item1 [name="Cheese", total=NULL, items=3, unit_cost=200]
> item_bill item1
600
> print item1
[name="Cheese",total=600,items=3,unit_cost=200]
```

eval <expr>

Name `eval`

Kind Command

Arg syntax <expression:any>

Description Parses the characters following it as an FTL expression and evaluates it. As with other commands the line is subject to $-expansion first.

Returns The value of the FTL expression

Example

```
> eval f="foibles"
"foibles"
> eval [a]:{echo a!; a} "asparagus"!
asparagus
"asparagus"
> set demo [expr]:{
>    io.write io.out ""+(expr)+" = "+(str (eval expr!)!)+"\n"!
> }
> forall <3..5> [n]:{demo "100/"+(str n!)!}
100/3 = 33
100/4 = 25
100/5 = 20
```

every <n> <command>

Name `every`

Kind Command

Arg syntax <milliseconds:int> <commands:restofline>

Description Executes the command line <restofline> every `milliseconds` forever or until the value returned by the command is first FALSE.
(Note that the command line is not an FTL expression.)
The function `sleep` is implicitly invoked by this command.
See also: `sleep`

If the number of milliseconds is negative the sleep function is not invoked.
(This may have a different effect to the sleep function being called with the argument zero.)

Returns Does not return

Example

```
> every 1000 echo tick
tick
tick
tick
tick
tick
tick
tick
tick
:

> set n 5
> set f[]:{n=n-1; io.fprintf io.out "%d\n"<n>!; n _gt_ 0}
> every -1 f
4
3
2
1
0
>
```

exit

Name `exit`

Kind Function

Arg syntax no argument

Description By closing all input streams on the current input stack this function normally causes execution in the interpreter to come to an end.

Returns Does not return

Example
```
> exit
> set error[msg]:{ echo msg!; exit! }
```

FALSE

Name `FALSE`

Kind Value

Arg syntax (no args)

Description Provides the `FALSE` value which can be used in boolean expressions.
Note that the value used for `FALSE` behaves as if it were defined as follows:

```
[code]:{FALSE}
```

See also: `TRUE`, the section about Boolean Values above.

Returns A value that is different to the one used for `TRUE`.

Example
```
> set done FALSE
> set once[fn]: { if (done) { done=TRUE; fn! } {}! }
{   rc = fn!;
    if (rc.error_occured) {echo "ERROR"!; FALSE} {TRUE}!
}
> eval FALSE {echo "never run"!}!
FALSE
```

for <dir> <closure>

Name `for`

Kind Function

Arg syntax <set:clodir> <enumerate:clocode>

Description This function executes `enumerate` for every name value pair in `set`.
If enumerate is code it is simply executed in the environment where `for` was invoked once for every member in the set, otherwise `enumerate` should be a closure with one argument.
If it is a closure with one argument, that argument is successively bound to different values in the name-value pairs in the `set`.

Returns NULL

Example

```
> for <..3> { echo "next"! }
next
next
next
> for <"one", "fine", "day"> echo
one
fine
day
```

forall <dir> <closure>

Name `forall`

Kind Function

Arg syntax <set:clodir> <enumerate:closure>

Description This function executes `enumerate` for every name value pair in `set`. `enumerate` should be a closure with two arguments. The first argument is successively bound to different values, while the second is bound to the corresponding name.
The order of evaluation is defined only for vectors.

Returns NULL

Example

```
set sieve[n]: {
    forall <0, n .. (len prime!)> [i]:{
        prime.(i) = NULL
    }!
}
> forall [a="ay", b="bee", c="see"] [pron, let]:{
>     echo "$let is pronounced '$pron'"!
> }
a is pronounced 'ay'
b is pronounced 'bee'
c is pronounced 'see'
> forall ([a,b]::{b=20+(a)} 1 2) [val, name]:{echo "$name =
$val"!}
b = 2
a = 1
```

forallwhile <dir> <closure>

Name `forallwhile`

Kind Function

Arg syntax <vals:clodir> <enumerate:closure>

Description This function executes `enumerate` for every name value pair, in order, in `vals` until the value returned by enumerate is not FALSE.
Enumerate must be a closure with exactly two unbound variables. The first

argument is successively bound to different values, while the second is bound to the corresponding name. If the execution returns NULL (e.g. leaves no value) the enumerations will not stop.

Returns TRUE if all the values in the sequence were used and FALSE otherwise.

Example

```
> set lastlessthan[seq,max]:{
    .ix = NULL;
    forallwhile seq [val,index]:{
        val _lt_ max {
            ix = index;
            TRUE
        }!
    }!;
    ix
}
> lastlessthan <14,88,142,156,188> 150
2
```

forwhile <dir> <closure>

Name `forallwhile`

Kind Function

Arg syntax <vals:clodir> <enumerate:clocode>

Description This function executes `enumerate` for every name value pair, in order, in `vals` until the value returned by enumerate is not FALSE. If the execution returns NULL (e.g. leaves no value) the enumerations will not stop. Enumerate must be a closure with exactly one unbound variables which is successively bound to each value in `vals`.

Returns TRUE if all the values in the sequence were used and FALSE otherwise.

Example

```
> set list [cmds]:{
    forwhile cmds [cmd]:{
        if cmd != "stop" {echo cmd!}{FALSE}!
    }!
}
> list <>
> list <"add","subtract","print">
add
subtract
print
TRUE
> list <"add","subtract","stop","ignore","ignore">
add
subtract
FALSE
```

func

Name `func`

Kind Command

Arg syntax <name:token> <value:closure>

Description Provide a name for the given closure in the current environment and mark it for automatic execution. If the name does not appear in the current environment it is added (to the environment with innermost-scope). The name may specify the directly in which it is set explicitly (by using the '`<dir>.<name>`' syntax). The innermost-scope can be specified explicitly using the syntax '`.<name>`'.

See also: `set`

Returns NULL

Example

```
> set noauto[a,b]:{a+b}
> func auto[a,b]:{a+b}
> eval noauto 1
[b]:{a+b}
> eval auto 1
[b]:{a+b}
> eval noauto 1 2
[]:{a+b}
> eval auto 1 2
3
> eval noauto 1 2 !
3
> eval auto 1 2 !
ftl $*console*:+39 in
ftl $*console*:40: a int value is not executable:
3
ftl $*console*:+39: failed to evaluate expression
> eval &auto 1 2
[]:{a+b}
> eval &auto 1 2 !
3
```

help

Name `help [all] [<command>]`

Kind Command

Arg syntax [all] [<command>]

Description If no `<command>` is provided on the command line this will print information about the commands in the top level environment. If `<command>` is provided help is provided for `<command>` explicitly.
Because named environments appearing at the front of a command are entered the syntax "`<environment> help`" can be used as an idiom that prints help for objects in `<environment>`.
For names associated with a directory this command indicates that this idiom

can be used to obtain more help or, if `all` is given, it recursively requests help for each name in the directory.
For names associated with a closure the command will print a string value associated with the name "`_help`".
Other types of value the command prints only the type of the value held.
These, however, are only included in the output if "`all`" is given on the command line.

Returns NULL

Example

```
> help
help [all] - prints command information
set <name> <expr> - set value in environment
exit - abandon all command inputs
:
sys help - show subcommands
parse help - show subcommands
io help - show subcommands
:
echo <whole line> - prints the line out
eval <expr> - evaluate expression
sleep <n> - sleep for <n> milliseconds
len [<env>|<closure>|<string>] - number of elements in object
> sys help
run <line> - execute system <line>
uid <user> - return the UID of the named user
time - system calendar time in seconds
localtime <time> - broken down local time
utctime <time> - broken down UTC time
localtimef <format> <time> - formatted local time
utctimef <format> <time> - formatted UTC time
> enter [ e=echo, h=help, x=exit, o=1, t=2 ]
> help
e <whole line> - prints the line out
h [all] [<cmd>] - prints information about commands
x - abandon all command inputs
> help all
e <whole line> - prints the line out
h [all] [<cmd>] - prints information about commands
x - abandon all command inputs
o - int value
t - int value
> help exit
exit - abandon all command inputs
> sys help time
time - system calendar time in seconds
> help sys.time
sys.time - system calendar time in seconds
> enter [fn = [_help="- my function"]:{echo _help!}, env =
[x="variable"], var = "variable" ]
> help
fn - my function
env help - show subcommands
```

```
> help all
fn - my function
env <subcommand> - commands:
    x - string value
var - string value
> env help
> env help all
x - string value
```

if <bool> <then-code> <else-code>

Name `if`

Kind Function

Arg syntax <test:bool> <then:code> <else:code>

Description Compares `test` with `FALSE` and then executes `then` if `test` was not `FALSE` and `else` otherwise.

Returns Returns the value returned by the execution of `then` or `else`, depending on which was selected for execution.

Example

```
> if 3 lt (rnd 6!) { echo "heads"! } { echo "tails"! }
> eval echo (if 3 lt (rnd 6!) {"heads"}{"tails"}!)!
set doit[cmd]: {
    if 0 == (cmp cmd "exit"!) {
       FALSE
    }{
       eval cmd!;
       TRUE
    }!
}
> if TRUE (echo "this") (echo "that")
this
> if FALSE (echo "this") (echo "that")
that
```

inenv <env> <name>

Name `inenv`

Kind Function

Arg syntax <env:clodir> <name>

Description returns 0 unless string <name> has a value in <env>

Returns NULL

Example

```
> inenv parse.env "inenv"
TRUE
> inenv [a=NULL, b] "a"
TRUE
> inenv [a=NULL, b] "b"
FALSE
> inenv [a=NULL, b] "c"
FALSE
```

invert <bool>

Name `invert`

Kind Function

Arg syntax <b:int>

Description Takes a booleans and returns the `TRUE` if the first is `FALSE`, or otherwise returns `FALSE`.

Returns Either the value `TRUE` or `FALSE`. When the argument is boolean the result is a boolean representing the not of `b`.
See also: `TRUE`, `FALSE`, `if`, `and`, `invert`

Example

```
> invert TRUE
FALSE
> invert FALSE
TRUE
```

io binfile <filename> <rw>

Name `io.binfile`

Kind Function

Arg syntax <filename:string> <access:string>

Description opens a binary file from the local filing system with name filename to which the requested access is provided. The access string consists of one or more of the following characters:

- "r" - open for read access
- "w" - open for write access

Note that the binding "`io.binfile <filename>`" can be used as a generic object to represent something openable as a stream.

On some operating systems this call differs from `io.file` in the handling of certain characters. All of the characters in the file are provided without translation when `io.binfile` is used.

Once opened the resulting stream should eventually be closed using `io.close`.

See also: `io.file`, `io.pathfile`, `io.close`

Returns NULL if the file could not be opened with the requested access otherwise a stream giving access (via `io.write` etc.) to the file

Example

```
> set exists[f]: { 0 != (cmp NULL (io.binfile f "r"!)!) }
> exists "/tmp/gone"
FALSE
> exists "ftl"
TRUE
> parse exec "" (io.binfile "script" "r"!)
```

io close <stream>

Name `io.close`

Kind Function

Arg syntax <open:stream>

Description Completes access to the given stream and makes it unavailable for future access

Returns NULL

Example

```
> io close io.file "test" "w"!
set log[msg]:{
    logstr = io.file "log" "w"!;
    io.write logstr msg!;
    io.write logstr "\n"!;
    io.close logstr!;
}
```

io err

Name `io.err`

Kind Read only value

Arg syntax Not a function

Description The standard stream to use for error output. (The same stream that is associated with `stderr` in C when the interpreter was started.)
See variables `io out` and `io in` too.

Returns Not a function

Example
```
> io write io.err "Wrong!\n"
7
set error[rc, msg]: {
    io.write io.err ""+(str rc!)+" "+(msg)+"\n"!
}
```

io file <filename> <rw>

Name `io.file`

Kind Function

Arg syntax <filename:string> <access:string>

Description opens a file from the local filing system with name filename to which the requested access is provided. The access string consists of one or more of the following characters:

- "r" - open for read access
- "w" - open for write access

Note that the binding "`file <filename>`" can be used as a generic object to represent something openable as a stream.

On some operating systems this call differs from `io.binfile` in the handling of certain characters. For example, on Windows, a control-Z character in the file is treated as an end-of-file character, preventing the reading of subsequent characters.

Once opened the resulting stream should eventually be closed using `io.close`.

See also: `io.file`, `io.pathfile`, `io.close`

Returns NULL if the file could not be opened with the requested access otherwise a stream giving access (via `io.write` etc.) to the file

Example
```
> set exists[f]: { 0 != (cmp NULL (io.file f "r"!)!) }
> exists "/tmp/gone"
FALSE
> exists "ftl"
TRUE
> parse exec "" (io.file "script" "r"!)
```

io fprintf <stream> <format> <env>

Name `io.fprintf`

Kind Function

Arg syntax <out:stream> <format:string> <args:clodir>

Description Writes the octets (bytes) of the characters generated by the formatted interpretation of the arguments in args according to the given format to the output stream provided. The format and arguments are interpreted as specified by the `strf` function.
The stream must have write access.
See also: `strf`, `io.write`

Returns The number of bytes written to the stream.

Example
```
> io fprintf io.out "€1 = £%[pounds]d.%[pennies]02d\n"
[pennies=60, pounds=0]
€1 = £0.60
14
> set printf io.fprintf io.out
> set n printf "sequence %02d %02d %02d %02d\n" <5..8>!
sequence 05 06 07 08
> eval n
21
```

io getc

Name `io.getc`

Kind Function

Arg syntax <in:stream>

Description Reads the next octet from the argument stream as a string. If no more characters are available from the stream (e.g. at the end of a file) then a NULL is given.
The stream must be open for reading.
See also: `io.read`

Returns A string containing the next octet read from the given stream.

Example

```
> set sin io.instring "In" "r"!
> io getc sin
"I"
> io getc sin
"n"
> io getc sin
>
set rev[st]:{
    .out="";
    .ch=NULL;
    while {NULL != (ch=io.getc st!)} {
        out=""+(ch)+(out);
    }!;
    out
}
> set sin io.instring "able was I ere" "r"!
> rev sin
"ere I saw elba"
```

io in

Name `io.in`

Kind Read only value

Arg syntax Not a function

Description The standard stream to use for normal input. (The same stream that is associated with `stdin` in C when the interpreter was started.)
See variables `io.out` and `io.err` too.

Returns Not a function

Example

```
> io in
<-'<stdin>'
set is_stdin[str]: { 0 == (cmp str io.in!) }
```

io instring <string> <rw>

Name `io.instring`

Kind Function

Arg syntax <input:string> <access:string>

Description creates a stream from a string for reading from. The access string consists of one or more of the following characters:

- "r" - open for read access
- "w" - open for write access

However this type of string can only be opened for reading. Nonetheless the binding "io.instring <string>" can be used as a generic object to represent something openable as a stream.

Returns NULL if the string could not be opened with the requested access otherwise a stream giving access (via `parse.exec` etc.) to the file

Example
```
> parse exec "" (io.instring "echo hi\necho there" "r"!)
hi
there
> set f io.instring "text to read\n" "r"!
> io read f 4
"text"
> io close f
```

io out

Name `io.out`

Kind Value

Arg syntax Not a function

Description The standard stream to use for normal output. (The same stream that is associated with `stdout` in C when the interpreter was started.)
See also: variables `io.err` and `io.in`

Returns Not a function

Example
```
> io out
<-EOF->
set myecho[msg]: {
    io.write io.out msg!;
    io.write io.out "\n"!;
}
```

io outstring <closure>

Name `io.outstring`

Kind Function

Arg syntax <code:closure>

Description Executes the given code with a stream argument that when written to creates a string. The string created is returned as a result of the function.

Returns The string generated by writes in the closure

Example

```
> set s io.outstring [out]:{ for <..5> {io.write out "line\
n"!}! }!
> echo $s
line
line
line
line
line
> set mystrf[fmt,args]:{io.outstring [out]:{io.fprintf out fmt
args!}!}
> mystrf "hi - %s" <"you">
"hi - you"
```

io pathfile <path> <filename> <rw>

Name `io.pathfile`

Kind Function

Arg syntax <path:string> <filename:string> <access:string>

Description opens a file from the local filing system using the given directory path with name filename and to which the requested access is provided. The directory path <path> consists of a series of directory names separated by the path separator character, which is ";" on Windows and ":" on other platforms. The access string consists of one or more of the following characters:

- "r" - open for read access
- "w" - open for write access

A file name is created from the concatenation of each directory on the <path>, the operating system's file separator character (`sys.fs.sep`), and the <filename>. An attempt to open each file name in turn is made until either an attempt is successful or no further directories are found.

Note that the binding "`pathfile <path> <filename>`" can be used as a generic object to represent something openable as a stream.

Once opened the resulting stream should eventually be closed using `io.close`.

See also: `sys.fs`, `io.file`, `io.close`

Returns `NULL` if the file could not be opened with the requested access in one of the directories given in the path otherwise a stream giving access (via `io.write` etc.) to a file.

Example

```
set include [name]:{
    .rdf = NULL;
    if (inenv sys.env "MYPATH"!) {
        rdf = io.pathfile sys.env.MYPATH name "r"!;
    }{  rdf = io.file name "r"!;
    }!;
    if (equal NULL rdf!) { echo "can't read file "+(str
name!)!; } {
        .ret = parse.exec "" rdf!;
        io.close rdf!;
        ret
    }!
}
> include "hello"
hello world
> set sys.env.MYPATH "."
> include "hello"
hello world
> set sys.env.MYPATH ".."
> include "hello"
can't read file "hello"
```

io read

Name `io.read`

Kind Function

Arg syntax <in:stream> <octets:integer>

Description Reads up to the given number of octets from the argument stream as a string. If no more characters are available from the stream (e.g. at the end of a file) then a string of length 0 is given. If there are characters available then at least one will be returned.
The stream must be open for reading.
See also: `io.read`

Returns A string containing at least one character read from the given stream and up to the given number, unless there are no more characters available, in which case an empy string is returned.

Example

```
> set f io.file "/tmp/1" "r"!
> io read f 50
"     Logs from Test inherited_option\n\n\nTest: inher"
```

io stringify <stream> <expr>

Name `io.stringify`

Kind Function

Arg syntax <out:stream> <expr:any>

Description Writes the expression `expr` to the output stream in a syntax (where possible) that would be parsed to re-create the expression value.
This function is equivalent to

```
  [stream, val]:{io.write stream (str val!)!}
```

The stream must have write access.
See also: `io.fprintf`, `io.write`, `str`

Returns The number of characters written to the output stream

Example

```
> set n io.stringify io.out "that's \"£££\"s\n"!
"that\'s \"\xc2\xa3\xc2\xa3\xc2\xa3\"s\n">
> eval n
41
> io stringify io.out [a=3, b="this", c=<"one", 2, 3>]
["a"=3,"b"="this","c"=<"one", 2, 3>]36
```

io write <stream> <string>

Name `io.write`

Kind Function

Arg syntax <out:stream> <text:string>

Description Writes the octets (bytes) of the characters in text to the output stream provided.
The stream must have write access.

Returns The number of bytes written to the stream.

Example

```
>io write io.out "€1 = £0.60\n"
€1 = £0.60
14
> set n io.write io.out "that's \"£££\"s\n"!
that's "£££"s
> eval n
17
```

joinchr <delim> <str>

Name `joinchr`

Kind Function

Arg syntax <delim:strintnl> <vec:clodir>

Description Join a vector of characters and strings separated by delimiters.
This function behaves the same way as join, but interprets integers in the delimiter or in the vector as a character ordinal number from the local "wide" character set.
Each character ordinal number is converted into a string of bytes representing the identified character.
See also: `join` and `split`.

Returns returns a string concatenating the items in the vector separated by the delimiter

Example
```
> eval echo (joinchr 167 <"para1", "para2", "para3">!)!
para1§para2§para3
> eval echo (joinchr NULL <915, 961, 945, 953>!)!
Γραι
> set chshow [n]:{ echo (joinchr " " <n .. 32+(n)>!)! }
> chshow 913
Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ ▫ Σ Τ Υ Φ Χ Ψ Ω Ϊ Ÿ ά έ ή ί ΰ
α
```

join <delim> <str>

Name `join`

Kind Function

Arg syntax <delim:strintnl> <vec:clodir>

Description Join a vector of octets and strings separated by delimiters.
If the delimiter is NULL the items in the vector are simply concatenated.
Any numbers occurring in the delimiter or in the vector are converted into octets (bytes) that are incorporated into the string. Such numbers are truncated to the normal octet range (0 to 255).
See also: `joinchr` and `split`.

Returns returns a string concatenating the items in the vector separated by the delimiter

Example
```
> join "/" <"dir", "subdir", "file">
"dir/subdir/file"
> join 0 <"line1", "line2">
"line1\0line2"
> join NULL <0x41, " line", 0x320, "made from", 32, "bytes">
"A line made from bytes"
```

leave

Name `leave`

Kind Function

Arg syntax No argument

Description This command pops the last argument directory pushed using either `enter` or `restrict` onto the current environment stack thus removing all the names in their directories from scope.
Since the commands `parse.local` and `help` are active only on the top directory in the current environment the command has a direct effect on them.
Note that (unless left explicitly using `leave`) the environment entered during the execution of a block goes out of scope at the end of the block, together with any pushed directories.
See also: `enter`, `restrict`, `leaving`

Returns NULL

Example

```
> enter [h=help, e=echo, ans=42]
> help
ans - int value
e <whole line> - prints the line out
h - prints command information
> eval h
["_help"="- prints command information"]::<func:0x8053dfc,0>
> eval help
["_help"="- prints command information"]::<func:0x8053dfc,0>
>
> set f [rec]:{ x="original"; enter rec!; echo x!; }
> f [x="new"]
new
> f [y="new"]
original
>
> eval ans
42
> set f1[]:{ans=80;ans}
> f1
80
> eval ans
80
> set f1[]:{enter [ans=9]!;ans}
> f1
9
> eval ans
80
>
> set item_bill[purchase]: {
    enter purchase!;
        total = items * unit_cost;
    leave!;
    purchase.total
};
> set item1 [name="Cheese", total=NULL, items=3, unit_cost=200]
> item_bill item1
600
> print item1
[name="Cheese",total=600,items=3,unit_cost=200]
```

leaving

Name `leaving`

Kind Function

Arg syntax No argument

Description This command pops the last argument directory pushed using either `enter` or `restrict` onto the current environment stack thus removing all the names in

their directories from scope. It then returns the environment removed in that way. (i.e. it is identical to `leave` except it returns the environment just left). Since the commands `parse.local` and `help` are active only on the top directory in the current environment the command has a direct effect on them. Note that (unless left explicitly using `leave` or `leaving`) the environment entered during the execution of a block goes out of scope at the end of the block, together with any pushed directories.
See also: `enter`, `leave`, `restrict`

Returns <env:clodir>

Example
```
> enter []
> set this 8
> leaving
[this=8]
```

len [<env>|<closure>|<string>]

Name `len`

Kind Function

Arg syntax <obj:clodirstr>

Description Evaluates the number of elements in the directory, closure or string.
The number of elements in a directory is the same as the number of name-value pairs in it.
The number of elements in a closure is also the same as the number of name-value pairs in it - any unbound names are not included and the code value is not involved.
The number of elements in a string is the number of bytes it contains.
See also: `strlen`.

Returns The number of elements in the argument

Example
```
> len "one\0two"
7
> len [a=5,b,c]
1
> len <..4>
4
> len "£"
2
> set stack <>
> set push [x]: { stack.(len stack!) = x; }
> push "this"
> push "that"
> eval stack
<"this", "that">
```

less <val> <val>

Name less

Kind Function

Arg syntax <val1:any> <val2:any>

Description Compares its first argument with the second and returns TRUE iff the first is strictly less than the second and FALSE otherwise. The comparison is made as described in the cmp function.
This function is identical to:

```
[val1, val2]: { 0 lt (cmp val1 val2!) }
```

See also:cmp, equal, notequal, more, moreeq, lesseq
Normally this function is associated with the lt operator.

Returns TRUE or FALSE

Example

```
set min[a,b]:{ if (less a b!){a}{b}! }
```

lesseq <val> <val>

Name lesseq

Kind Function

Arg syntax <val1:any> <val2:any>

Description Compares its first argument with the second and returns TRUE iff the first is less than, or equal to, the second and FALSE otherwise. The comparison is made as described in the cmp function.
This function is identical to:

```
[val1, val2]: { 0 le (cmp val1 val2!) }
```

See also:cmp, equal, more, lesss, moreeq, lesseq
Normally this function is associated with the le operator.

Returns TRUE or FALSE

Example

```
set inrange[x,low,high]: { 1 and (moreeq x low!) and (lesseq x
high!) }
```

lock <env>

Name lock

Kind Function

Arg syntax <env:clodir>

Description Prevents the addition or removal of names from the directory provided. It does not prevent the modification of values associated with existing names. There is no associated `unlock` function.
When a closure is locked the unbound variables can still be bound.
See also: `new`.

Returns NULL

Example
```
> set milk [mon=4, tue=2, wed=4, thu=3, fri=3, sat=6, sun=0]
> lock milk
> set milk.mon 5
> eval milk.mon
5
> set milk.feb 4
ftl $*console*:6: can't set a value for "feb" here
> set config_empty []
> lock config_empty
```

logand <bool> <bool>

Name `logand`

Kind Function

Arg syntax <b1:bool> <b2:bool>

Description Takes two booleans and returns the `FALSE` if the first is `FALSE`, or otherwise returns the second.

Returns Either the value `FALSE` or the second argument. When both arguments are boolean the result is a boolean representing the anded combination of `b1` and `b2`.
See also: `TRUE`, `FALSE`, `if`, `logor`, `invert`, `bitand`

Example
```
> logand TRUE TRUE
TRUE
> logand TRUE FALSE
FALSE
> logand FALSE TRUE
FALSE
> logand FALSE FALSE
FALSE
```

logor <bool> <bool>

Name `logor`

Kind Function

Arg syntax <b1:bool> <b2:bool>

Description Takes two booleans and returns the `TRUE` if the first is not `FALSE`, or otherwise returns the second.

Returns Either the value TRUE or the second argument. When both arguments are boolean the result is a boolean representing the ored combination of `b1` and `b2`.
See also: TRUE, FALSE, `if`, `logand`, `invert`, `bitor`

Example
```
> logor TRUE TRUE
TRUE
> logor TRUE FALSE
TRUE
> logor FALSE TRUE
TRUE
> logor FALSE FALSE
TRUE
```

more <val> <val>

Name `more`

Kind Function

Arg syntax <val1:any> <val2:any>

Description Compares its first argument with the second and returns TRUE iff the first is strictly more than the second and FALSE otherwise. The comparison is made as described in the `cmp` function.
This function is identical to:

```
[val1, val2]: { 0 gt (cmp val1 val2!) }
```

See also:`cmp`, `equal`, `notequal`, `less`, `moreeq`, `lesseq`
Normally this function is associated with the `gt` operator.

Returns TRUE or FALSE

Example
```
set max[a,b]:{ if (more a b!){a}{b}! }
```

moreeq <val> <val>

Name `moreeq`

Kind Function

Arg syntax <val1:any> <val2:any>

Description Compares its first argument with the second and returns TRUE iff the first is more than, or equal to, the second and FALSE otherwise. The comparison is made as described in the `cmp` function.
This function is identical to:

```
[val1, val2]: { 0 ge (cmp val1 val2!) }
```

See also:`cmp`, `equal`, `notequal`, `more`, `less`, `lesseq`
Normally this function is associated with the `ge` operator.

Returns TRUE or FALSE

Example
```
set inrange[x,low,high]: { (moreeq x low!) and (lesseq x high!)
}
```

mul <int> <int>

Name `mul`

Kind Function

Arg syntax <n1:int> <n2:int>

Description Takes two integers and multiplies them together.

Returns An integer with the value n1*n2.
See also: `add`, `sub`, `div`, `shiftl`, `shiftr, neg`

Example
```
> mul 7 24
168
```

neg <int>

Name `neg`

Kind Function

Arg syntax <n:int>

Description Takes an integer and returns its negated value.

Returns An integer with the value -n.
See also: `add`, `sub`, `mul`, `shiftl`, `shiftr`

Example
```
> neg 10
-10
> neg -10
10
```

new <env>

Name `new`

Kind Function

Arg syntax <env:clodir>

Description Creates a new copy of the argument made by copying each name-value pair in the argument. Note, though, that it does not make copies of the values copied.

Bug: does not copy unbound variables, but does copy code body in closure

Returns NULL

Example

```
> set base [val=1]
> set copy1 base
> set copy2 (new base!)
> set base.val 2
> eval copy1
[val=2]
> eval copy2
[val=1]
>
> set class [val=0]:{ [read=[]:{val}, write=[n]:{val=n;}] }
> set rw1 class!
> set rw2 class!
> eval rw1.write 42!
> eval rw2.read!
42
> # oops!
>
> set class [val=0]:{ [read=[]:{val}, write=[n]:{val=n;}] }
> set rw1 (new class!)!
> set rw2 (new class!)!
> eval rw1.write 42!
> eval rw2.read!
0
> eval rw1.read!
42
```

notequal <val> <val>

Name `notequal`

Kind Function

Arg syntax <val1:any> <val2:any>

Description Compares its first argument with the second and returns TRUE iff they are not equal and FALSE otherwise. The comparison is made as described in the `cmp` function.
This function is identical to

```
[val1, val2]: { 0 != (cmp val1 val2!) }
```

See also:`cmp`, `equal`, `more`, `less`, `moreeq`, `lesseq`
Normally this function is associated with the `!=` operator.

Returns TRUE or FALSE

Example

```
{   if (notequal var NULL!) {
        do_something_with var!
    }{}!
}
```

NULL

Name NULL

Kind Value

Arg syntax (no args)

Description Provides access to the only value in the "nul" type. This can be used to provide a value that is guaranteed not to be equal to values of other types.

Returns string of (multi-octet) characters with ordinal `widechar`

Example

```
> set newobj [name1=NULL, name2=NULL]
> set lastval NULL
> set val[n]: { lastval=n }
> cmp NULL ({}!)
0
{   rc = fn!;
    if (rc.error_occured) {NULL} {rc.answer}!
}
```

octet <int>

Name `octet`

Kind Function

Arg syntax <byte:int>

Description Returns single byte string containing the given octet
See also: `chr`

Returns A string of `len` 1 whose only octet has the ordinal number provided.

Example

```
> echo Wow${octet 46!}
Wow.
> octet 0
"\0"
```

octetcode <char>

Name `octetcode`

Kind Function

Arg syntax <char:string>

Description Returns the ordinal position of the first byte (octet) in its argument in the current locale's character set.

Returns An integer containing the ordinal number.
See also: `octet`, `chrcode`

Example

```
> eval "£"
"\xc2\xa3"
> octetcode "\xc2\xa3"
164
> set dump[a]:{
>     forall (split NULL a!) [x]:{
>         io.write io.out " "+(strf "%02x" (octetcode x!)!)!
>     }!;
>     io.write io.out "\n"!;
> }
> dump "hexadecimal"
 68 65 78 61 64 65 63 69 6d 61 6c
```

parse argv

Name `parse.argv`

Kind Directory

Arg syntax Not a function

Description A read-only vector of command line argument strings. The value with name 0 is the command, as invoked. (Compare with `parse.codeid`)

Returns Not a function

Example

```
% ftl -- one two
:
> eval parse.argv
[0="ftl", 1="one", 2="two"]
/usr/bin/ftl -- --myopt val
:
> eval parse.codeid!
"ftl"
> eval parse.argv.0
"/usr/bin/ftl"
```

parse codeid

Name `parse.codeid`

Kind Function

Arg syntax None

Description Returns an identification assigned to the interpreter by the invoker.

Returns A string containing the name of the interpreter assigned by the library-using application

Example

```
> parse codeid
"ftl"
> set err[msg]:{
>     io.write io.err ""+(parse.codeid!)+": "+(msg)+"\n"!;
> }
> err "something wrong"
ftl: something wrong
```

parse env

Name `parse.env`

Kind Function

Arg syntax None

Description When run this function returns a directory containing all the names and values in scope at the point of execution.
See also: `restrict`, `parse.root`, `parse.local`

Returns The current invocation environment.

Example

```
> join " " (sortdom (parse.env!)!)
"FALSE NULL TRUE chr chrcode cmd cmp collate do domain echo
enter equal eval every exit fmt forall help if inenv int io ip
join
joinchr len less lesseq lock mac more moreeq new notequal octet
octetcode parse range restrict return rnd set sleep sort sortby
sortdom sortdomby source sourcetext split str strf strlen sys
tolower toupp
er type typename typeof while"
> set extra [val1=1, val2=2]:{ domain (parse.local!)! }
> extra
<"val2", "val1", "extra", "push", "stack", "len", "sleep",
"eval",
"echo", "cmd", "every", "sortdomby", "sortby", "sortdom",
"sort",
"range", "domain", "restrict", ...
```

parse errors

Name `parse.errors`

Kind Function

Arg syntax (no argument)

Description Provides an indication of the number of errors so far encountered. If this number is greater than zero most interpreters will pass back a non-zero return code to their execution environment.
Also see `parse.errors_reset` and `parse.newerror`.

Returns total number of errors encountered by parser

Example

```
> parse errors
5
if 0 == (parse.errors!) { } {
    echo "Parse failed with "+(str (parse.errors!)!)+"errors"!
}!
```

parse errors_reset <errors>

Name `parse.errors_reset`

Kind Function

Arg syntax <errors:int>

Description Resets the number of errors so far encountered to `errors`. If this number is greater than zero most interpreters will pass back a non-zero return code to their execution environment.
Also see `parse.errors` and `parse.newerror`.

Returns total number of errors encountered by parser

Example

```
> parse errors_reset
> parse errors
0
{   saved = parse.errors!;
    dodgy_function!;
    parse.errors_reset saved!;
}!
```

parse exec <cmds> <stream>

Name `parse.exec`

Kind Function

Arg syntax <cmds:string> <in:stream>

Description Executes the commands in the given string and then reads further commands from the given stream and executes them, returning once the commands have been read. The value generated by the last command is then returned.
If the intitial commands string is `NULL` it is ignored.
If the stream is `NULL` (e.g. unopened) it is ignored.
See also: `source`

Returns The value returned by the last command executed.

Example

```
> set script_in io.instring "e restricted; e script; x" "r"!
> parse exec "restrict [e=echo, x=exit]" script_in
restricted
script
> sys run cat /tmp/1
"Temporary script"
> set run[script]:{ parse.exec "" (io.file script "r"!)! }
> set result run "/tmp/1"!
"Temporary script"
> eval result
"Temporary script"
> parse exec "echo command line" NULL
command line
```

parse local

Name `parse.local`

Kind Function

Arg syntax None

Description When run this function returns a directory containing all the names and values in scope locally. (That is in the most recently established environment forming part of the execution environment).
See also: `parse.root`, `parse.env`

Returns The local invocation environment (in which local variables are created).

Example

```
> set extra [domain=domain, parse=parse]::{ domain
(parse.local!)! }
> extra
<"parse", "domain">
> set extra [val1=1, val2=2]:{ domain (parse.env!)! }
> extra
<"val1", "val2", "help", "set", "exit", "forall", "if",
"while",
"do", "sys", "parse", "io", "source", "sourcetext", "return",
"typeof", "typename", "cmp", ...
```

parse line

Name `parse.line`

Kind Function

Arg syntax None

Description Returns the line number in the current source of characters being interpreted. The character source used is the one that was selected at the first character of the last command executed.
See also: `parse.source`

Returns The current line number.

Example

```
> parse line
27
> set thisline[]:{ echo "line "+(str (parse.line!)!)! }
> parse exec "echo init; thisline" (io.instring "thisline"
"r"!)
init
line 1
line 2
```

parse newerror

Name `parse.newerror`

Kind Function

Arg syntax (no argument)

Description Adds one to the number of errors registered encountered by the parser. Also see `parse.errors` and `parse.errors_reset`

Returns NULL

Example

```
> set myerror[msg] {
    echo msg!;
    parse.newerror!;
}
> parse errors
17
> parse newerror
> parse errors
18
```

parse assoc

Name `parse.assoc`

Kind Directory

Arg syntax Not a function

Description This directory names the different kinds of operator associativity that are meaningful in operator definitions. The values are integers that can be used in operator definitions. The names are strings of up to three characters constructed:
`<left-hand associativity>f<right-hand associativity>`
in which a prefix operator has no left hand associativity, a postfix operator has no right hand associativity but an infix operator has both. Where it exists the associativity is specified either as "x" or "y". An "x" indicates that no association is possible: any argument on the relevant side must have higher precidence. A "y" indicates that other operators at the same precidence are possible on that side. In consequence a series of "xfy" operators will associate to the left:
a xfy b xfy c = (a xfy b) xfy c
and a series of "yfx" operators will associate to the right:
a yfx b yfx b = a yfx (b yfx b)
Also see `parse.op`, `parse.opset` and `parse.scanop`

Returns Not a function

Example

```
> domain parse.assoc
<"yfy", "yfx", "xfy", "xfx", "xf", "yf", "fx", "fy">
> parse opset parse.op 3 parse.assoc.xfx "eq" equal
> eval 9 eq 9
TRUE
> # xfx is non-associative
> eval 9 eq 7 eq 5
ftl $*console*:116: undefined symbol 'eq'
ftl $*console*:116: trailing text in expression: ...5
ftl $*console*:116: warning - trailing text '5'
> set assocstr[as]:{ (domain (select (equal as)
parse.assoc!)!).0 }
> assocstr 2
"fy"
> assocstr parse.assoc.xfy
"xfy"
```

parse op

Name `parse.op`

Kind Directory

Arg syntax Not a function

Description This directory is the definition of a set of operators. This specific directory is one used by the default parser, and so opertor definitions that appear in it will be part of the language.

Such definitions are numerically indexed from zero contiguously. The index determines the relative precidence between operators. Each entry is itself a directory of name-value pairs in which the names are operator names and the values are directories with values for the names "`assoc`" and "`fn`". The

associativity and arity of the operator is determined by the value given to "`assoc`" and may take one of the values held in `parse.assoc` (with meaning described there). The function that is executed to represent the operator is given to "`fn`". This will be a closure with two unbound variables if the operator is diadic, and with one unbound variable if it is monadic.

Although the function `parse.opset` can be used to create new entries in this directory, the data can also be inspected, rewritten or altered by the user in any other way.

Also see `parse.assoc`, `parse.opset` and `parse.scanop`

Returns Not a function

Example

```
> set forops[opdefs, fn]:{
>     forall parse.op [precfns, prec]:{
>         forall precfns [precfn, name]:{
>             fn name prec precfn.assoc precfn.fn!
>         }!
>     }!
> }
> set assocstr[as]:{ (domain (select (equal as)
parse.assoc!)!).0 }
> set printf io.fprintf io.out
> forops parse.op [name, prec, assoc, fn]:{
>     printf "%3d: %3s %v\n" <prec, assocstr assoc!, name>!;
> }
  0: xfy "or"
  1: xfy "and"
  2:  fy "not"
  3: xfx "=="
  3: xfx "!="
  3: xfx "lt"
  3: xfx "le"
  3: xfx "gt"
  3: xfx "ge"
  4: xfy "shl"
  4: xfy "shr"
  5:  fy "-"
  6: xfy "+"
  6: xfy "-"
  7: xfy "*"
  7: xfy "/"
> # ignore operators from now on:
> set parse.op NULL
> eval parse.op.4.shr
["fn"=["_help"="<n1> <n2> - return n1 right shifted by n2
bits", "_1", "_2"]::
<func:0x805c5d9,2>, "assoc"=7]
> eval parse.op.4.shr.fn 16 2!
4
> eval 16 shr 2
4
```

parse opset <opdefs> <prec> <assoc> <name> <function>

Name `parse.opset`

Kind Function

Arg syntax <opdefs:dir> <prec:int> <assoc:int> <name:string> <function:closure>

Description This function modifies the operator definition `opdefs` so that it includes or replaces an entry that defines a new operator with the given name as having the precidence indicated by `prec`, the associativity indicated by `assoc`, and which will be implemented by the given `function`.
The associativity must be one of the integers held by `parse.assoc`.
The directory created will be have entries for separate precidences as indicated by the `prec` argument. Note, however, that when the definitions come to be used (e.g. in `parse.opeval`) a continuous range of precidences from zero upward is expected. For this reason it may be necessary to apply the definitions to the `range` function before they are used.
This function can be used to prepare a new set of operation definitions for `parse.op`.
Also see `parse.assoc`, `parse.op` and `parse.scanop`

Returns NULL

Example

```
> set ops <>           # op definitions with explicit
priorities
>
> parse opset ops 6  parse.assoc."fy"  "_ng_"  neg
> parse opset ops 8  parse.assoc."xfy" "_sb_"  sub
> parse opset ops 8  parse.assoc."yfx" "_rsb_" [x,y]:{y-x}  #
reverse subtract
> parse opset ops 10 parse.assoc."fy"  "_mi_"  neg
>
> set opdef range ops! # op definitions with relative
priorities
>
> parse opeval opdef {20 _sb_ 4}
16
> parse opeval opdef {15 _rsb_ 30}
15
> parse opeval opdef {20 _rsb_ 15 _rsb_ 30} # 20 _rsb_ (15
_rsb_ 30)
-5
> parse opeval opdef {20 _sb_ 4 _sb_ 6}     # (20 _sb_ 4) _sb_
6
10
> parse opeval opdef {_ng_ 20 _sb_ 4}       # _ng_ (20 _sb_ 4)
-16
> parse opeval opdef {_mi_ 20 _sb_ 4}       # (_mi_ 20) _sb_ 4
-24
```

parse opeval <opdefs> <code>

Name `parse.opeval <opdefs> <code>`

Kind Function

Arg syntax <opdefs:dir> <code:code>

Description This function uses the operator definitions held in opdefs to evaluate the expression held in the given code value.
Also see `parse.assoc`, `parse.op`, `parse.opset` and `parse.scanop`

Returns The result of evaluating the expression.

Example

```
> set ops <>      # op definitions with explicit priorities
> set opdef NULL  # op definitions with relative priorities
>
> set opset[pri, as, name, fn]:{
>     parse.opset ops pri parse.assoc.(as) name fn!;
>     opdef = range ops!;
> }
>
> # unquoted value format
> set fmt.t [f,p,v]:{ if (equal "string" (typename v!)!) {v}
{str v!}!}
>
> set diadic[strf=strf,name,l,r]::{ strf "(%s %t %t)" <name, l,
r>! }
> set monadic[strf=strf,name,lr]::{ strf "(%s %t)" <name,
lr>! }
> opset 6 "fy" "fy" (monadic "fy")
> opset 6 "fx" "fx" (monadic "fx")
> opset 6 "yfy" "yfy" (diadic "yfy")
> opset 6 "xfx" "xfx" (diadic "xfx")
> opset 6 "xfy" "xfy" (diadic "xfy")
> opset 6 "yfx" "yfx" (diadic "yfx")
> opset 6 "yf" "yf" (monadic "yf")
> opset 6 "xf" "xf" (monadic "xf")
> opset 8 "fy" "Fy" (monadic "Fy")
> opset 8 "fx" "Fx" (monadic "Fx")
> opset 8 "yfy" "yFy" (diadic "yFy")
> opset 8 "xfx" "xFx" (diadic "xFx")
> opset 8 "xfy" "xFy" (diadic "xFy")
> opset 8 "yfx" "yFx" (diadic "yFx")
> opset 8 "yf" "yF" (monadic "yF")
> opset 8 "xf" "xF" (monadic "xF")
>
> set opeval[expr]: { parse.opeval opdef expr! }
>
> opeval {4}
4
> opeval {"str"}
"str"
> opeval {<3>}
<3>
```

```
> opeval {(4)}
4
> opeval {5 xfy 6}
"(xfy 5 6)"
> opeval {5 yfx 6}
"(yfx 5 6)"
> opeval {5 yfy 6}
"(yfy 5 6)"
> opeval {5 xfx 6}
"(xfx 5 6)"
> opeval {fx 7}
"(fx 7)"
> opeval {fy 7}
"(fy 7)"
> opeval {4 xf}
"(xf 4)"
> opeval {4 yf}
"(yf 4)"
> opeval {5 xfy 6 xfy 7}
"(xfy (xfy 5 6) 7)"
> opeval {5 yfx 6 yfx 7}
"(yfx 5 (yfx 6 7))"
>
> opeval {3-4}
ftl $*console*:131: code has trailing text: ...-4
```

parse root

Name `parse.root`

Kind Function

Arg syntax None

Description When run, this function returns a directory containing all the names and values in the directory in scope at the start of execution. Note that additions or deletions to the directory may have been made, but none of the values held in argument directories or `entered` directories will be visible.
As with other functions access to this command can be eliminated using the `restrict` function.
See also: `parse.env`, `parse.local` and `enter`.

Returns The root directory.

Example
```
> restrict [exit=exit, help=help, domain=domain, parse=parse,
enter=enter]
> enter [local=parse.local, env=parse.env, root=parse.root]
> help
local - return local current invocation directory
env - return current invocation environment
root - return current root environment
> domain (parse.local!)
```

```
<"local", "env", "root">
> domain (parse.env!)
<"local", "env", "root", "exit", "help", "domain", "parse",
"enter">
> domain (parse.root!)
<"help", "set", "exit", "forall", "if", "while", "do", "sys",
"parse", "io",
"source", "sourcetext", "return", "typeof", "typename", "cmp",
"type", "NULL",
"rnd", "int", "TRUE", "FALSE", "equal", "notequal", "less",
"lesseq", "more",
"moreeq", "fmt", "str", "strf", "collate", "toupper",
"tolower", "strlen",
"octet", "chr", "octetcode", "chrcode", "split", "join",
"joinchr", "ip",
"mac", "new", "lock", "inenv", "enter", "restrict", "domain",
"range", "sort",
"sortdom", "sortby", "sortdomby", "every", "cmd", "echo",
"eval", "sleep",
"len">
>
> # leaving parse.root available gives people access to all the
commands:
> domain (myrun=(parse.root!).sys.run)
<"_help">
> myrun pwd
/home/cgg/tree/clean/v5/src
```

parse scan <cmd>

Name `parse.scan`

Kind Function

Arg syntax <cmd:string>

Description This function prepares a string for parsing by subsequent parse functions. It returns a parse object which contains the part of the string that has not yet been parsed. In general, successful parse functions consume the initial part of this string and update the object. The string remaining can be inspected using the parse.scanned function.
See also: `parse.scanned`, `scan<others>`

Returns A parse object which can be used as an argument to other `parse.scan`* functions.

Example

```
> set p parse.scan "123 identifier"!
> parse scanned p
"123 identifier"
> parse scanint @n p
TRUE
> parse scanned p
" identifier"
```

parse scanned <parseobj>

Name parse.scanned

Kind Function

Arg syntax <parseobj:clodir>

Description This function expects a parse object as an argument that was initially created by parse.scan. It is used to retrieve the part of string remaining to be parsed.
See also: parse.scan, scan<others>

Returns This string contained by a parse object that remains to be consumed by parse functions.

Example

```
> set p parse.scan "123 identifier"!
> parse scanned p
"123 identifier"
> parse scanint @n p
TRUE
> parse scanned p
" identifier"
```

parse scanempty <parseobj>

Name parse.scanempty

Kind Function

Arg syntax <parseobj:clodir>

Description This function expects a parse object as an argument that was initially created by parse.scan and which contains the remainder of the string to be parsed. If it is an empty string, with no characters in it. It will return TRUE, otherwise it returns FALSE.
The intended use of this function is to ensure that the existing parsing of a string is complete - and to ensure that there is no trailing text.
See also: parse.scan, parse.scanned, scan<others>

Returns TRUE if an empty string was parsed, FALSE otherwise.

Example

parse scanwhite <parseobj>

Name parse.scanwhite

Kind Function

Arg syntax <parseobj:clodir>

Description This function expects a parse object as an argument that was initially created by `parse.scan` and which contains the remainder of the string to be parsed.
If the string begins with white space characters (spaces, newlines, tabs etc.) it will return TRUE and update the parse object with the remainder of the string, otherwise it returns FALSE.
The intended use of this function is to skip over white space that may separate other elements to parse.
See also: `parse.scan`, `parse.scanned`, `scan<others>`

Returns TRUE if one or more white space characters were parsed, FALSE otherwise.

Example

parse scanspace <parseobj>

Name `parse.scanspace`

Kind Function

Arg syntax <parseobj:clodir>

Description This function expects a parse object as an argument that was initially created by `parse.scan` and which contains the remainder of the string to be parsed. If the string begins with white space characters (spaces, newlines, tabs etc.) it will update parseobj.0 with the remainder of the string. It will always return TRUE.
The intended use of this function is to skip over semantically insignificant white space that may separate other elements to parse.
See also: `parse.scan`, `parse.scanned`, `scan<others>`

Returns TRUE.

Example

parse scanint <@int> <parseobj>

Name `parse.scanwint`

Kind Function

Arg syntax <ref_int:closure> <parseobj:clodir>

Description This function expects a parse object as its last argument that was initially created by `parse.scan` and which contains the remainder of the string to be parsed. If the string begins with an integer it will return TRUE and update `parseobj.0` with the remainder of the string, otherwise it returns FALSE. If the parse succeeds the value of the hexadecimal parsed is applied to the `ref_int` closure as an integer.
The intended use of ref_int is to provide a destination for the parsed quantity in the form of a reference (e.g. "`@name`"), however any closure requiring a single argument can be used.
See also: `parse.scan`, `parse.scanned`, `scan<others>`

Returns TRUE if a number is parsed, FALSE otherwise.

Example

```
> set numid[p,ref_n,ref_id]:{
> (parse.scanint ref_n p)! (parse.scanwhite p)!
> (parse.scanid ref_id p)! (parse.scanempty p)!
> }
> numid (parse.scan "7 baloons"!) @howmany @what
TRUE
> eval howmany
7
> eval what
"baloons"
```

parse scanhex <@int> <parseobj>

Name parse.scanhex

Kind Function

Arg syntax <ref_int:closure> <parseobj:clodir>

Description This function expects a parse object as its last argument that was initially created by parse.scan and which contains the remainder of the string to be parsed. If the string begins with a hexadecimal integer it will return TRUE and update parseobj.0 with the remainder of the string, otherwise it returns FALSE. If the parse succeeds the value of the hexadecimal parsed is applied to the ref_int closure as an integer.
The intended use of ref_int is to provide a destination for the parsed quantity in the form of a reference (e.g. "@name"), however any closure requiring a single argument can be used.
See also: parse.scan, parse.scanned, scan<others>

Returns TRUE if a hexadecimal number were parsed, FALSE otherwise.

Example

parse scanhexw <width> <@int> <parseobj>

Name parse.scanhexw

Kind Function

Arg syntax <width:int> <ref_int:closure> <parseobj:clodir>

Description TThis function expects a parse object its last argument that was initially created by `parse.scan` and which contains the remainder of the string to be parsed. If the string begins with exactly `width` hexadecimal characters it will return `TRUE` and update `parseobj.0` with the remainder of the string, otherwise it returns `FALSE`. If the parse succeeds the value of the hexadecimal parsed is applied to the `ref_int` closure as an integer.
The intended use of ref_int is to provide a destination for the parsed quantity in the form of a reference (e.g. "`@name`"), however any closure requiring a single argument can be used.
See also: `parse.scan`, `parse.scanned`, `scan<others>`

Returns `TRUE` if `width` hexadecimal characters were parsed, `FALSE` otherwise.

Example

parse scanstr <@string> <parseobj>

Name `parse.scanstr`

Kind Function

Arg syntax <ref_string:closure> <parseobj:clodir>

Description This function expects a parse object as its last argument that was initially created by `parse.scan` and which contains the remainder of the string to be parsed. If the string begins with a quoted string it will return `TRUE` and update `parseobj.0` with the remainder of the string, otherwise it returns `FALSE`. If the parse succeeds the characters of the parsed string applied to the `ref_string` closure as a string.
The intended use of ref_string is to provide a destination for the parsed quantity in the form of a reference (e.g. "`@name`"), however any closure requiring a single argument can be used.
See also: `parse.scan`, `parse.scanned`, `scan<others>`

Returns TRUE if a string is parsed, FALSE otherwise.

Example

parse scanid <@string> <parseobj>

Name `parse.scanid`

Kind Function

Arg syntax <ref_string:closure> <parseobj:clodir>

Description This function expects a parse object as its last argument that was initially created by `parse.scan` and which contains the remainder of the string to be parsed. If the string begins with a identifier (that is an alphabetic character (or '_') followed by a number of alphanumeric (or '_') characters) it will return `TRUE` and update `parseobj.0` with the remainder of the string, otherwise it returns `FALSE`. If the parse succeeds the characters of the parsed identifier applied to the `ref_string` closure as a string.
The intended use of ref_string is to provide a destination for the parsed quantity in the form of a reference (e.g. "`@name`"), however any closure requiring a single argument can be used.
See also: `parse.scan`, `parse.scanned`, `scan<others>`

Returns TRUE if an identifier is parsed, FALSE otherwise.

Example

```
> set numid[p,ref_n,ref_id]:{
> (parse.scanint ref_n p)! (parse.scanwhite p)!
> (parse.scanid ref_id p)! (parse.scanempty p)!
> }
> numid (parse.scan "7 baloons"!) @howmany @what
TRUE
> eval howmany
7
> eval what
"baloons"
```

parse scanitem <delims> <@string> <parseobj>

Name `parse.scanitem`

Kind Function

Arg syntax <delims:string> <ref_string:closure> <parseobj:clodir>

Description This function expects a parse object as its last argument that was initially created by `parse.scan` and which contains the remainder of the string to be parsed. If the string begins with a sequence of non-delimiter characters (delimiter characters are those from the string `delims`) it will return `TRUE` and update `parseobj.0` with the remainder of the string, otherwise it returns `FALSE`. If the parse succeeds the characters of the parsed item applied to the `ref_string` closure as a string.
The intended use of ref_string is to provide a destination for the parsed quantity in the form of a reference (e.g. "`@name`"), however any closure requiring a single argument can be used.
See also: `parse.scan`, `parse.scanned`, `scan<others>`

Returns TRUE if one or more non-delimiter characters were parsed, FALSE otherwise.

Example

parse scanitemstr <@string> <parseobj>

Name parse.scanitemstr

Kind Function

Arg syntax <ref_string:closure> <parseobj:clodir>

Description This function expects a parse object as an argument that was initially created by parse.scan and which contains the remainder of the string to be parsed. If the string begins with either a sequence of non-white characters or a quoted string it will return TRUE and update parseobj.0 with the remainder of the string, otherwise it returns FALSE. If the parse succeeds a characters of the parsed item or string are applied to the ref_string closure.
The intended use of ref_string is to provide a destination for the parsed quantity in the form of a reference (e.g. "@name"), however any closure requiring a single argument can be used.
See also: parse.scan, parse.scanned, scan<others>

Returns TRUE if an item or a quoted string is parsed, FALSE otherwise.

Example

parse scanmatch <prefixes> <@string> <parseobj>

Name parse.scanmatch

Kind Function

Arg syntax <prefixes:dir> <ref_val:closure> <parseobj:clodir>

Description This function expects a parse object as an argument that was initially created by parse.scan and which contains the remainder of the string to be parsed. The first argument is a directory of name-value pairs. This function attempts to parse one of the names as a prefix in the string. If the string begins one of the names it will return TRUE and update parseobj.0 with the remainder of the string, otherwise it returns FALSE. If the parse succeeds the value associated with the parsed name in prefixes is applied to the ref_val closure.
The intended use of ref_string is to provide a destination for the parsed quantity in the form of a reference (e.g. "@name"), however any closure requiring a single argument can be used.
See also: parse.scan, parse.scanned, scan<others>

Returns TRUE if an prefix contained in delims is parsed, FALSE otherwise.

Example

```
> set cmd "a long line\n"
> set translations [a=5, line="LINE", long=<99>]
> set toparse parse.scan cmd!
> set tran NULL
> parse scanmatch translations @tran toparse
TRUE
> parse scanned toparse
" long line\n"
> eval tran
5
> parse scanspace toparse
TRUE
> parse scanmatch translations @tran toparse
TRUE
> parse scanned toparse
" line\n"
> eval tran
<99>
> parse scanspace toparse
TRUE
> parse scanmatch translations @tran toparse
TRUE
> parse scanned toparse
"\n"
> eval tran
"LINE"
```

parse scanops <baseparse> <opdefs> <@string> <parseobj>

Name `parse.scanops`

Kind Function

Arg syntax <baseparse:closure> <opdefs:dir> <ref_val:closure> <parseobj:clodir>

Description This function expects a parse object as its last argument that was initially created by `parse.scan` and which contains the remainder of the string to be parsed.
The first argument must be a parsing closure that has two unbound arguments: the first returning a value ("@val") and the second containing a directory like the one provided as the first argument to this function.
The second argument is a directory of operator definitions such as that constructed by `parse.opset`. The base values connected by these operators are to be parsed by the `baseparse` argument.
This function attempts to parse an expression in the string constructed with the operators defined in `opdefs`. If the string begins with such an expression it will return `TRUE` and update `parseobj.0` with the remainder of the string, otherwise it returns `FALSE`. If the parse succeeds the value associated with the parsed expression is applied to the `ref_val` closure.
The intended use of ref_string is to provide a destination for the parsed quantity in the form of a reference (e.g. "`@name`"), however any closure requiring a single argument can be used.

See also: `parse.scan`, `parse.scanned`, `scan<others>`, `parse.scan<others>`, `parse.opset`

`THIS FUNCTION IS NOT YET IMPLEMENTED`

Returns `TRUE` if a prefix contained in `delims` is parsed, `FALSE` otherwise.

Example

parse source

Name `parse.source`

Kind Function

Arg syntax None

Description Returns the name of the current source of characters being interpreted. The name is the name of the source as it was at the first character of the last command executed.
See also: `parse.line`

Returns The current interpreted source.

Example

```
> parse source
"$*console*"
> set thissrc[]:{ echo "source "+(str (parse.source!)!)! }
> parse exec "echo init; thissrc" (io.instring "echo start\n
thissrc" "r"!)
init
source "\$<INIT>"
start
source "\$<string-in>"
```

range <env>

Name `range`

Kind Function

Arg syntax <mapping:dir>

Description Constructs a vector containing the values that are named in the mapping provided. Values associated with names that are hidden because of scoping rules will not appear. The order of entries in the vector is the same as those that are generated in the `domain` function. If the mapping is itself a vector (numerically indexed) the order of entries will be identical to the mapping, otherwise the order is undefined with respect to the mapping.
See also: `domain`

Returns A vector of the values contained in the argument directory,

Example

```
> range [mon="eggs", tue="bacon", wed="flour"]
<"eggs", "bacon", "flour">
> echo ${join " == " (range [a="£3", b="€4.5", "c"="$$4"]!)!}
£3 == €4.5 == $4
```

restrict <env>

Name `restrict`

Kind Function

Arg syntax <env:dir>

Description restrict further commands to those in <env>

Returns This command removes the current environment stack from scope and pushes its argument directory `env` onto the current environment stack thus bringing all the names defined in the directory into scope.
Since the commands `parse.local` and `help` are active only on the top directory in the current environment the command has a direct effect on them.
Note that the environment entered during the execution of a block goes out of scope at the end of the block, together with any pushed directories. This will mean that the original environment will become available again once the block incorporating the `restrict` has exited.
This function can be used to reduce the complexity of the syntax available subsequently. It can be particularly useful as an argument to `parse.exec` in order to constrain a file's syntax. Note, however, that including any functions with one or more arguments among those allowed may enable the current environment to be altered (using the <name> = <value> FTL syntax) - this may be overcome by defining commands to replace the functions using the `cmd` function. The `lock` function may be of use if this is a problem.
See also: `enter`, `leave`, `lock`, `cmd`, `parse.exec`

Example

```
> restrict [h=help, e=echo, x=leave, ans=42]
> h
h [all] [<cmd>] - prints information about commands
e <whole line> - prints the line out
x - exit the environment last entered or restricted
> eval h            # we don't have an 'eval' command any more
ftl $*console*:+27 in
ftl $*console*:28: unknown command 'eval h'
> x
> eval h            # we have eval again, but not 'eval'
ftl $*console*:+29 in
ftl $*console*:30: undefined symbol 'h'
ftl $*console*:+29: failed to evaluate expression
% cat > /tmp/2
e simple
e script
x
^D
% SCRIPT=/tmp/2 ftl
```

```
ftl: v1.6
> set script_in io.file sys.env.SCRIPT "r"!
> parse exec "restrict [e=echo, x=exit]" script_in
simple
script
```

return <rc>

Name `return`

Kind Function

Arg syntax <rc:int>

Description abandon current command input returning <rc>

Returns A string containing the name of the interpreter assigned by the library-using application

Example

```
> set script_in io.instring "echo starting; return 2; echo
unreached" "r"!
> set rc parse.exec "" script_in!
starting
2
> eval rc
2
```

rnd <n>

Name `rnd`

Kind Function

Arg syntax <n:int>

Description Choose pseudo random number less than n and greater or equal to zero.

The random number generator produces results based on a fixed sequence of random numbers. The sequence of numbers produced is determinstic and will depend on the initial value of a "seed".

See also: `rndseed`

Returns A random integer

Example

```
> echo ${rnd 256!}.${rnd 256!}.${rnd 256!}.${rnd 256!}
4.62.35.205
> set choose[vec]:{ vec.(rnd (len vec!)!) }
> choose <"red", "white", "blue">
"blue"
```

rndseed <n>

Name `rndseed`

Kind Function

Arg syntax <seed:int> or <seed:string>

Description Set the "seed" used for the random number generator.
Up to 32-bits of entropy inferred from the seed which then determines the sequence of values returned from the `rnd` function. The value given can either be in the form of an integer or a binary string. There may be several different values of seed that will result in the same sequence of values returned from `rnd`. Whenever the specific seed is provided the same sequence results, though.

If, for example, the random number sequence is used to build test data that same test data can be regenerated in a subsequent run by recording and re-establishing the same rndseed prior to execution.

The sequence of values returned for a given seed is dependent on the build of the interpreter. It is likely, for example, that the same seed will yield different sequences when used on interpreters used on different types of computer or operating system.

See also: `rnd, sys.osfamily`

Returns NULL

Example

```
> set test[]:{
>     io.fprintf io.out "%d.%d.%d.%d colour " <
>                       rnd 256!, rnd 256!,
>                       rnd 256!, rnd 256! >!;
>     echo <"red", "white", "blue">.(rnd 3!)!;
> }
> set myseed sys.time!; echo Using seed $myseed
Using seed 1555250522
> rndseed myseed
> test
245.32.252.87 colour blue
> test
128.151.116.88 colour white
> rndseed myseed
> test
245.32.252.87 colour blue

> rndseed "a longer string to extract a seed from"
```

select <closure> <dir>

Name `select`

Kind Function

Arg syntax <choose:clocode> <set:clodir>

Description This function executes `choose` for every name-value pair in `set` and returns the subset of set for which `choose` returns TRUE.
If `choose` is a code value it is simply executed in the environment where `select` was invoked, once for every member in the set, otherwise `choose` may be a closure with either one or two arguments.
If it is a `choose` with one argument, that argument is successively bound to different values in the name-value pairs in the `set`.
If it is a closure with two arguments, the first argument is successively bound to different values, while the second is bound to the corresponding name
The order of evaluation is defined only for vectors.
See also: TRUE

Returns The subset of `set` for which `choose` returned TRUE as a directory.

Example

```
> eval select [v]:{ equal "int" (typename v!)! }
<1,2,"this",4,"that">!
<1, 2, 3=4>
> eval select [v]:{ equal "string" (typename v!)! }
<1,2,"this",4,"that">!
<2="this", 4="that">
> set inorder select [max=0,v]:{ if (more v max!) { max=v; TRUE
} {FALSE}! }
> inorder <"ay", "bee", "see", "dee", "ee">
<"ay", "bee", "see">
> inorder <2,3,4,2,4,0,7>
<2, 3, 4, 6=7>
> set start[n, vec]: { select [v,ix]:{ moreeq ix n! } vec! }
> start 3 <"metis", "adrasthea", "amalthea", "thebe", "io",
"europa">
<3="thebe", "io", "europa">
> set pints [mon=1, tue=3, wed=2, thu=2, fri=6, sat=0, sun=0]
> select (lesseq 2) pints
["tue"=3, "wed"=2, "thu"=2, "fri"=6]
> select (equal 0) pints
["sat"=0, "sun"=0]
```

set <name> <expr>

Name `set`

Kind Command

Arg syntax <name:token> <value:any>

Description Provide a name for the given value in the current environment. If the name does not appear in the current environment it is added (to the environment with innermost-scope). The name may specify the directly in which it is set explicitly (by using the '`<dir>.<name>`' syntax). The innermost-scope can be specified explicitly using the syntax '`.<name>`'.
This command provides a similar function to the FTL expression `<name> = <value>`.

See also: `func`

Returns NULL

Example
```
> set name "string value"
> set dir [old=0]
> set dir.old "new"
> set dir.(dir.old) 1
> eval dir
["old"="new","new"=1]
> set proc dir:{ echo old! }
> proc
new
> set fn [old]:{ echo old! }
> fn "this"
this
```

shiftl <int> <int>

Name `shiftl`

Kind Function

Arg syntax <n1:int> <n2:int>

Description Takes two integers and shifts the bit representation of the first left by the number of bits specified in the second, filling newly vacated bit positions with zeros.
This is normally equivalent to dividing n1 by 2 to the power of n2.
Shifts by a negative amount leave the value unchanged.

Returns An integer with the value n1 shifted left by n2.
See also: `add`, `sub`, `mul`, `div`, `shiftl`, `shiftr, neg`

Example
```
> shiftl 1 3
8
> shiftl 1 -3
8
> shiftl 1 (-3)
1
```

shiftr <int> <int>

Name `shiftr`

Kind Function

Arg syntax <n1:int> <n2:int>

Description Takes two integers and shifts the bit representation of the first left by the number of bits specified in the second, filling newly vacated bit positions with zeros.
For a positive n1, this is normally equivalent to dividing n1 by 2 to the power of n2. This is not true for negative numbers however.
Shifts by negative amounts leave the value unchanged.

Returns An integer with the value n1 shifted left by n2.
See also: `add`, `sub`, `mul`, `div`, `shiftl`, `shiftr, neg`

Example
```
> shiftr 16 3
2
> shiftr 16 (-3)
16
> shiftr -16 3
2305843009213693950
```

sleep <n>

Name `sleep`

Kind Function

Arg syntax <milliseconds:int>

Description Sleep for the given number of milliseconds.
A C function can be delegated to the parser to execute during this call using the C function `parser_suspend_put()`. Such functions can perform background processing during the wait.
See also: `every`

Returns NULL

Example
```
> sleep 500
>
> set rpt[ms,fn]:{ while {1} {fn!; sleep ms!;}! }
> rpt 1000 { echo "tick"! }
tick
tick
tick
tick
tick
tick
```

sort <env>

Name `sort`

Kind Function

Arg syntax <env:clodir>

Description Provides a vector of names in the provided environment which is ordered by the values each name is associated with in the environment.
This function is identical to:

```
sortby cmp
```

(but is implemented more efficiently).
See also: `sortby`, `sortdom`, `cmp`, `range`

Returns A vector of names from the given environment with entries for each of the name-value pairs in it, in which entries are ordered so that the directory values of names with a smaller index are less than or equal to those with a larger index according to the comparison function `cmp`.

Example

```
> sort <-9,NULL,[a=3],"string",45,<>,3>
<1, 0, 6, 4, 3, 2, 5>
> set max[vec]:{ vec.((sort vec!).(-1+(len vec!))) }
> max <2,6,3,5,6,8,-8>
8
> set min[vec]:{ vec.((sort vec!).0) }
> min <2,6,3,5,6,8,-8>
-8
```

sortby <cmpfn> <env>

Name `sortby`

Kind Function

Arg syntax <cmpfn:closure> <env:clodir>

Description Provides a vector of names in the provided environment which is ordered by the values each name is associated with in the environment. The ordering is dictated by the `cmpfn` function which is executed with two arguments to be compared, and must return

- a negative integer value to indicate that the first argument is less than the second
- a zero integer value to indicate that the first argument is equal to the second
- a positive integer value to indicate that the first argument is more than the second

See also: `sort`, `sortdomby`, `range`

Returns A vector of names from the given environment with entries for each of the name-value pairs in it, in which entries are ordered so that the directory values of names with a smaller index are less than or equal to those with a larger index according to the comparison function provided.

Example

```
> sortby [a,b]:{cmp b a!} <-9,NULL,[a=3],"string",45,<>,3>
<2, 5, 3, 4, 6, 0, 1>
> set cmpmin [v1,v2]:{ cmp v1.((sort v1!).0) v2.((sort v2!).0)!
}
> set maxmin[vecvec]:{ sortby cmpmin vecvec! }
> maxmin <<0,2,3>,<8>,<9,7,-1>,<4,3>>
<2, 0, 3, 1>
```

sortdom <env>

Name `sortmod`

Kind Function

Arg syntax <env:clodir>

Description Provides a vector of names in the provided environment which is ordered by the names in the environment. Note that because the "names" in a vector are already in order this function is more useful on other kinds of directory.
This function is identical to:

```
sortmodby cmp
```

(but is implemented more efficiently).
See also: `sortmodby`, `sort`, `cmp`, `domain`

Returns A vector of names from the given environment with entries for each of the name-value pairs in it, in which entries are ordered so that the names with a smaller index are less than or equal to those with a larger index according to the comparison function `cmp`.

Example

```
> sortdom <4,5,67,2>
<0, 1, 2, 3>
> sortdom [one=1, two=2, three=3]
<"one", "three", "two">
> set in_order [_vals, _fn]:{
>     forall (sortdom _vals!) [_val]:{ _fn _vals.(_val)
_val! }!
> }
> in_order [apples=12, pears=8, bananas=7] [n,fruit]:{
>    io.fprintf io.out "we have %d %s\n" <n,fruit>!
> }
we have 12 apples
we have 7 bananas
we have 8 pears
```

sortdomby <cmpfn> <env>

Name `sortdomby`

Kind Function

Arg syntax <cmpfn:closure> <env:clodir>

Description Provides a vector of names in the provided environment which is ordered by the names in the environment. The ordering is dictated by the `cmpfn` function which is executed with two arguments to be compared, and must return

- a negative integer value to indicate that the first argument is less than the second
- a zero integer value to indicate that the first argument is equal to the second
- a positive integer value to indicate that the first argument is more than the second

See also: `sortdom`, `sortby`, `domain`

Returns A vector of names from the given environment with entries for each of the name-value pairs in it, in which entries are ordered so that the names with a smaller index are less than or equal to those with a larger index according to the comparison function provided.

Example

```
> sortdomby [a,b]:{cmp b a!} [one=1, two=2, three=3]
<"two", "three", "one">
> sortdomby collate ["fred"=7, "£"=99, "FORMER"=2, "104"=77]
<"104", "FORMER", "fred", "\xc2\xa3">
```

source <filename>

Name `source`

Kind Command

Arg syntax <tokstring:filename>

Description Open the filename for reading and interpret lines from that file as commands to be executed.
Normally file execution continues until it is completely read or until a `return` or `exit` command are executed.
When the last octet is read from the file, or the interpreter is closed, the file will be closed automatically.
See also: `parse.exec`, `sourcetext`, `command`, `return`, `exit`

Returns NULL

Example When c:\\extra-commands contains:

```
echo executing…
```

```
> source c:\\extra-commands
executing...
> set test[]:{ source "C:\\extra-commands"!; echo "finished"! }
> test
executing…
finished
>
```

sourcetext <stringexpr>

Name `sourcetext`

Kind Function

Arg syntax <text:stringorcode>

Description Execute the provided text (provided either as a string or a code value) as if it were source commands.

Normally execution continues until it is completely read or until a `return` or `exit` command are executed.
See also: `parse.exec`, `source`, `command`, `return`, `exit`

Returns Any value provided by a `return` command.

Example

```
> sourcetext "echo executing..."
executing...
> sourcetext {
>    echo executing…
>}
executing…

> set state "executing"
> sourcetext " enter [state=\"running\"]; echo ${state}...; "
executing...
> sourcetext { enter [state="running"]; echo ${state}...; }
running...
> sourcetext " enter [state=\"running\"]; echo \${state}...; "
running…

> set test[]:{ sourcetext "echo executing...\n"!; echo
"Started"! }
> test
Started

> set IF[cond,then,else]:{
>    if cond {sourcetext then!}{sourcetext else!}!
> }
> IF TRUE {
>     echo true
>     return 1
> } {
>     echo false
>     return 0
> }
true
1
```

split <delim> <str>

Name `split`

Kind Function

Arg syntax <delimiter:stringnl> <arg:string>

Description Splits arg into a vector of strings separated by the delimiter. If the delimiter is an emtpy string the argument is split into (possibly multi-byte) characters. If it is NULL then the argument is split into octets.
See also: `join`, `joinchr`, `len`, `strlen`

Returns A string containing the name of the interpreter assigned by the library-using application

Example

```
> split "/" "dir/subdir/file"
<"dir", "subdir", "file">
> split "\0" "line1\0line2"
<"line1", "line2">
> split "" "£€¢"
<"\xc2\xa3", "\xe2\x82\xac", "\xc2\xa2">
> split NULL "£€¢"
<"\xc2", "\xa3", "\xe2", "\x82", "\xac", "\xc2", "\xa2">
> split NULL "ascii"
<"a", "s", "c", "i", "i">
> set spaceout[s]: { join " - " (split "" s!)! }
> echo ${spaceout "£€¢"!}
£ - € - ¢
> set args[str]: { range (select (notequal "") (split " "
str!)!)! }
> args " a   badly  spaced     command line"
<"a", "badly", "spaced", "command", "line">
```

str <expr>

Name `str`

Kind Function

Arg syntax <value:any>

Description Generate a string representation of the value provided. The string representation is one that, if read as a value should (where possible) regenerate the value. This function is equivalent to

```
[val]: { io.outstring [out]:{ io.stringify out val! }! }
```

See also: `io.stringify`, `io.strf`

Returns A string containing the name of the interpreter assigned by the library-using application

Example

```
> set showstr[val]: { io.write io.out "value: "+(str val!)+"\
n"!; }
> set show[val]: { io.write io.out "value: "+(val)+"\n"!; }
> show "shaln't"
value: shaln't
> showstr "shaln't"
value: "shaln\'t"
> showstr 100
value: 100
> show 100
ftl: value has wrong type - type is int, expected string
ftl $*console*:49: string expected after '+'
ftl $*console*:49: error in '+(val)+"\n"!; '
["_1"=<-EOF->,"_help"="<stream> <string> - write string to
stream", "_2"]::<func:0x8053ff4,2>
> set s str "that's \"£££\"s\n"!
> echo $s
"that\'s \"\xc2\xa3\xc2\xa3\xc2\xa3\"s\n"
> eval s
"\"that\\\'s \\\"\\xc2\\xa3\\xc2\\xa3\\xc2\\xa3\\\"s\\n\""
> eval str [a=3, b="this", c=<"one", 2, 3>, d=<7..99>]!
"[\"a\"=3,\"b\"=\"this\",\"c\"=<\"one\", 2, 3>,\"d\"=<7 .. 9>]"
```

strf <fmt> <env>

Name `strf`

Kind Function

Arg syntax <format:string> <values:clodir>

Description Generate a string according to the format provided from the values given in the values directory.
The format string must conform to the following syntax:
[<text> | %% | %[<fieldname>]<format>]*
where <text> is simply a sequence of non '%' characters, the optional <fieldname> is alphanumeric text enclosed by '[' and ']' and the <format> is [-][+][0][ ][#][<digits>+][.<digits>+][<fmtchar> | {<fmtstr>}]
in which the optional prefixes have these uses:

prefix	meaning
-	left justify the string in the specified width, otherwise right justify it.
+	include a sign (if relevant to the value) both when negative and when positive.
0	when right justified pad leading positions with zeros to accommodate the specified width (taking any place for a sign into account).
(space)	leave a blank in place of a positive sign (if relevant to the value) when positive

#	use an alternative form of the format.
<digits>+	specifies the minimum width of the field, more characters than this may be printed if the format requires it.
.<digits>+	specifies the "precision" of the format. This has a different interpretation in different formats. In floating point formats (none at present) it would specify a number of decimal digits following the decimal point. In many other formats it specifies the maximum field width (in which surplus characters generated by the format are discarded).

The format name is then specified either as a single character `<fmtchar>` or as multiple characters in braces `{<fmtstr>}`. This name is looked up in the "`fmt`" directory in the root environment to obtain a binding that is invoked in order to generate the basic output. The formats supported can be extended by including additional bindings in this directory. The bindings are invoked as iff called using

fmt.<name> <flags> <precision> <value>!

This location (`fmt`) may be altered in future versions.

Curently the following formats are supported (although the user can add more):

Format name	Value type required	String produced
v	any type	the string representation of the type (as generated by the `str` function)
s	string	the string
S	string	the whole string or the initial part that is terminated with a NUL character ("\0")
x	(unsigned) integer	lower case hexadecimal
X	(unsigned) integer	upper case hexadecimal
d	integer	decimal
u	(unsigned) integer	decimal
b	integer	binary encoded little endian
B	Integer	binary encoded big endian
j	any type	the JSON representation of the value[1]
J	any value	a "pretty printed" JSON representation of the[2] value (contains additional white space)

Each <text> is copied directly to the output string. The sequence %% copies a

1 Only when compiled with the JSON FTL extension
2 Only when compiled with the JSON FTL extension

single percent to the output string. Each <format> selects an argument from the given values and formats it according to the rules above. The argument selected will be the one indexed either by the <fieldname> if it is provided, or by the next integer (starting from an initial index of 0).

See also: `str`, `io.fprintf`

Returns A string containing the formatted representation of the arguments provided in the given environment.

Example

```
> set on_1line[val]:{io.write io.out (strf "%-.79v"
<val>!)!;io.write io.out "\n"!;}
> on_1line 3
3
> on_1line fmt
[d=[_1, _2, _3]::<func:0x55e8c0516e8d,3>, u=[_1, _2,
_3]::<func:0x55e8c0516eb9,

> fmt help
d <f> <p> <val> - %d integer format
u <f> <p> <val> - %u unsinged format
x <f> <p> <val> - %x hex format
X <f> <p> <val> - %X hex format
s <f> <p> <val> - %s string format
S <f> <p> <val> - %s zero terminated string format
c <f> <p> <val> - %c character format
b <f> <p> <val> - %b little endian binary format
B <f> <p> <val> - %B big endian binary format
v <f> <p> <val> - %v value format
j <f> <p> <val> - %j (JSON) value format
J <f> <p> <val> - %J (pretty JSON) value format

> set hdump[str]:{
>     forall (split NULL str!) [h]:{
>         io.write io.out (strf "%02X" h!)!;
>      }!
> }

> set printf io.fprintf io.out
> printf "tokens %[one]t (not %[one]v) and %[two]t\n"
[one="string", two=9]
tokens string (not "string") and 9
35

> set show[fmt,val]:{
>   .chrs=printf fmt <val>!;
>   printf "\n(%d characters)\n"<chrs>!;
> }
> show "%J" [a=10, b=<3,5,7>, c="this"]
{
    "a" : 10,
    "b" : [3, 5, 7],
    "c" : "this"
}
```

```
(57 characters)

> set packet[op,dest,data]:{
>    outstring [s]:{
>        printf "%.1b%.4b%s" <op, dest, data>!;
>    }!
> }
> packet 2 0x12131415 "somedata"
"\x02\x15\x14\x13\x12somedata"
```

Currently there is a bug in integer formats that occurs when the 0 and the + prefixes are provided together.

strlen [<string>]

Name strlen

Kind Function

Arg syntax <chars:string>

Description Gives the number of (possibly multibyte) characters in string. The argument string is interpreted as a sequence of multi-byte characters according to the current locale. Typically the UTF-8 encoding of the Unicode character set will be used. Because some characters are not encoded in a single octet this will not always give the same result as `len`.
The number of characters in a string will correspond to the number of elements in the result of `split ""`.
See also: `len`, `split`, `joinchr`

Returns The number of characters held in a string

Example
```
> len "£€¢"
7
> strlen "£€¢"
3
> strlen "ascii"
5
```

sub <int> <int>

Name sub

Kind Function

Arg syntax <n1:int> <n2:int>

Description Takes two integers and subtracts the second from the first.

Returns An integer with the value n1-n2.
See also: `add`, `mul`, `div`, `shiftl`, `shiftr, neg`

Example

```
> sub 0x100 12
244
> sys localtimef "%T" (sub (sys.time!) 600!)
"17:10:14"
```

sys env

Name `sys.env`

Kind Directory

Arg syntax Not a function

Description A non-enumerable directory containing the names and string values of the system environment. This directory can be both read from and written to, but must be indexed by strings and must be given string values.

Returns Not a function

Example

```
> sys env USER
"gray"
> set sys.env.new str [a=4, b=7]!
> eval ${sys.env.new}
["a"=4,"b"=7]
```

sys localtime <time>

Name `sys.localtime`

Kind Function

Arg syntax <time:int>

Description Takes a number of seconds representing a calendar time (e.g. as returned by sys.time) and breaks it down into components representing elements of the local time relative to the current timezone in a returned directory with fields:

- sec - number of seconds past the minute (0 .. 61 - normally up to 59)
- min - number of minutes pas the hour (0 .. 59)
- hour - number of hours into the identified date (0 .. 23)
- mday - day in the month (1 .. 31)
- mon - month in the year (1 .. 12)
- year - year number (e.g. 2006)
- wday - number of days since Sunday (0 .. 6)
- yday - number of days since 1 January (0 .. 365)
- isdst - whether daylight savings time is in force (=0 no, >0 yes, <0 unknown)

See also: `sys.localtimef`, `sys.utctime`, `sys.time`

Returns A directory with values for names `sec`, `min`, `hour`, `mday`, `mon`, `year`, `wday`, `yday` and `isdst`.

Example

```
> sys localtime (sys.time!)
["sec"=39, "min"=46, "hour"=12, "mday"=21, "mon"=5,
"year"=2006, "wday"=3,
"yday"=171, "isdst"=1]
set logrec[out, when, rec]:{
    t = sys.localtime when!;
    io.write out (octet 0+(t.year)-2000!)!;
    io.write out (octet t.mon!)!;
    io.write out (octet t.mday!)!;
    io.write out (octet t.hour!)!;
    io.write out (octet t.min!)!;
    io.write out (octet (len rec!)!)!;
    io.write out rec!;
}
```

sys localtimef <format> <time>

Name `sys.localtimef`

Kind Function

Arg syntax <format:string> <time:int>

Description Takes a number of seconds representing a calendar time (e.g. as returned by sys.time) and returns the local time (relative to the locally set timezone) formatted according to the given format string. Specific implementations may vary, but the format string should be interpreted as specified in ISO/IEC 9899:1999 (C programming language) clause 7.23.3.5.3 (strftime function format string) which includes the specification of these fields:

- %a is replaced by the locale’s abbreviated weekday name
- %A is replaced by the locale’s full weekday name
- %b is replaced by the locale’s abbreviated month name
- %B is replaced by the locale’s full month name
- %c is replaced by the locale’s appropriate date and time representation
- %C is replaced by the year divided by 100 and truncated to an integer, as a decimal number (00−99)
- %d is replaced by the day of the month as a decimal number (01−31)
- %D is equivalent to ‘‘%m/%d/%y’’
- %e is replaced by the day of the month as a decimal number (1−31); a single digit is preceded by a space
- %F is equivalent to ‘‘%Y−%m−%d’’ (the ISO 8601 date format)
- %g is replaced by the last 2 digits of the week-based year (see below) as a decimal number (00−99)
- %G is replaced by the week-based year (see below) as a decimal number (e.g., 1997)
- %h is equivalent to ‘‘%b’’
- %H is replaced by the hour (24-hour clock) as a decimal number (00−23)
- %I is replaced by the hour (12-hour clock) as a decimal number (01−12)
- %j is replaced by the day of the year as a decimal number (001−366)
- %m is replaced by the month as a decimal number (01−12)

- %M is replaced by the minute as a decimal number (00−59)
- %n is replaced by a new-line character
- %p is replaced by the locale’s equivalent of the AM/PM designations associated
- with a 12-hour clock
- %r is replaced by the locale’s 12-hour clock time
- %R is equivalent to ‘‘%H:%M’’
- %S is replaced by the second as a decimal number (00−60)
- %t is replaced by a horizontal-tab character.
- %T is equivalent to ‘‘%H:%M:%S’’ (the ISO 8601 time format)
- %u is replaced by the ISO 8601 weekday as a decimal number (1−7), where Monday is 1
- %U is replaced by the week number of the year (the first Sunday as the first day of week 1) as a decimal number (00−53)
- %V is replaced by the ISO 8601 week number (see below) as a decimal number (01−53)
- %w is replaced by the weekday as a decimal number (0−6), where Sunday is 0
- %W is replaced by the week number of the year (the first Monday as the first day of week 1) as a decimal number (00−53)
- %x is replaced by the locale’s appropriate date representation
- %X is replaced by the locale’s appropriate time representation
- %y is replaced by the last 2 digits of the year as a decimal number (00−99)
- %Y is replaced by the year as a decimal number (e.g., 1997)
- %z is replaced by the offset from UTC in the ISO 8601 format ‘‘−0430’’ (meaning 4 hours 30 minutes behind UTC, west of Greenwich), or by no characters if no time zone is determinable
- %Z is replaced by the locale’s time zone name or abbreviation, or by no characters if no time zone is determinable
- %% is replaced by %

See also: `sys.utctimef`, `sys.localtime`, `sys.time`

Returns A string containing the local time formatted as requested.

Example

```
> set timestr sys.localtimef "%H~%M"
> timestr (sys.time!)
"11~58"
```

sys osfamily

Name `sys.osfamily`

Kind Read only variable

Arg syntax Not a function

Description Returns a string classifying the category of operating system running. Currently supported possible values are:

- "linux"
- "windows"
- "sunos"

Returns Not a function.

Example

```
> sys osfamily
"linux"
set osfile[path]: {
    sep = if (equal sys.osfamily "windows"!) {"\\"}{"/"}!;
    join sep path!
}
> osfile <"dir","subdir","file">
"dir/subdir/file"
```

sys run <line>

Name `sys.run`

Kind Command

Arg syntax <line:string>

Description Executes the system command held on the line.

Returns NULL if the execution was successful otherwise a string describing the error

Example

```
> set rc sys.run "asfd"!
sh: asfd: command not found
> eval rc
"ftl \$*console*:63: system command failed - \'asfd\' (rc
32512)\n"
> sys run arp -a
? (10.17.20.254) at 00:06:5B:00:F9:60 [ether] on eth0
```

sys time

Name `sys.time`

Kind Function

Arg syntax No argument

Description Provides a representation of calendar time as a number of seconds since an operating system-dependent initial epoch.
See also: `sys.localtime`, `sys.localtimef`, `sys.utctime`, `sys.utctimef`

Returns An integer number of seconds.

Example

```
> sys localtimef "%X" (sys.time!)
"10:49:07"
> set timeit[fn]: {
>    start = sys.time!;
>    fn!;
>    0+(sys.time!)-(start)
> }
> timeit (sleep 2000)
2
```

sys uid <user>

Name `sys.uid`

Kind Command

Arg syntax <user:string>

Description return a unique identifier for the named user as an integer, or NULL if there is no such user.

Returns UID of a user.

Example

```
> sys uid gray
> set isroot[id]: { 0 == (cmp id (sys.uid "root"!)!) }
> isroot 500
FALSE
> isroot 0
TRUE
```

sys utctime <time>

Name `sys.utctime`

Kind Function

Arg syntax <time:int>

Description Takes a number of seconds representing a calendar time (e.g. as returned by sys.time) and breaks it down into components representing elements of the Coordinated Universal Time (UTC) in a returned directory with fields:

- sec - number of seconds past the minute (0 .. 61 - normally up to 59)
- min - number of minutes pas the hour (0 .. 59)
- hour - number of hours into the identified date (0 .. 23)
- mday - day in the month (1 .. 31)
- mon - month in the year (1 .. 12)
- year - year number (e.g. 2006)
- wday - number of days since Sunday (0 .. 6)
- yday - number of days since 1 January (0 .. 365)
- isdst - whether daylight savings time is in force (=0 no, >0 yes, <0 unknown)

See also: `sys.utctimef`, `sys.localtime`, `sys.time`

Returns A directory with values for names `sec`, `min`, `hour`, `mday`, `mon`, `year`, `wday`, `yday` and `isdst`.

Example

```
> sys utctime (sys.time!)
["sec"=41, "min"=47, "hour"=11, "mday"=21, "mon"=5,
"year"=2006,
"wday"=3, "yday"=171, "isdst"=0]
set logrec[out, when, rec]:{
    t = sys.utctime when!;
    io.write out (octet 0+(t.year)-2000!)!;
    io.write out (octet t.mon!)!;
    io.write out (octet t.mday!)!;
    io.write out (octet t.hour!)!;
    io.write out (octet t.min!)!;
    io.write out (octet (len rec!)!)!;
    io.write out rec!;
}
```

sys utctimef <format> <time>

Name `sys.utctimef`

Kind Function

Arg syntax <format:string> <time:int>

Description Takes a number of seconds representing a calendar time (e.g. as returned by sys.time) and returns the Coordinated Universal Time (UTC) formatted according to the given format string. Specific implementations may vary, but the format string should be interpreted as specified in ISO/IEC 9899:1999 (C programming language) clause 7.23.3.5.3 (strftime function format string) which includes the specification of the fields used by sys.localtimef.
See also: `sys.localtimef`, `sys.utctime`, `sys.time`

Returns A string containing the UTC time formatted as requested.

Example

```
> set stamp {sys.utctimef "%T%z" (sys.time!)!}
> stamp
"00:06:52+0000"
```

TRUE

Name TRUE

Kind Value

Arg syntax (no args)

Description Provides a TRUE value which can be used in boolean expressions.
Note that the value used for TRUE behaves as if it were defined as follows:

```
[code]:{code!}
```

See also: TRUE, the section about Boolean Values above.

Returns An value that is not the one used for FALSE.

Example
```
> set done FALSE
> set once[fn]: { if (done) { done=TRUE; fn! } {}! }
{   rc = fn!;
    if (rc.error_occured) {echo "ERROR"!; FALSE} {TRUE}!
}
```

type <typename>

Name `type`

Kind Command

Arg syntax <typename>

Description Returns a type value of the named type. Values of this type can be compared with each other.
See also: `typeof`, `typename`

Returns A type value

Example
```
> type string
string
> set isdir[x]: { 0 == (cmp (type{dir}!) (typeof x!)!) }
```

typename <expr>

Name `typename`

Kind Command

Arg syntax <expr:any>

Description Returns a string representing the type of the expression when it has been evaluated.
See also: `type`, `typeof`

Returns A type value

Example
```
> typeof "string"
string
> set isdir[x]: { 0 == (cmp "dir" (typename x!)!) }
```

typeof <expr>

Name `typename`

Kind Function

Arg syntax <expr:any>

Description Returns a type value representing the type that `expr` evaluates to. When displayed these types display as the following:

- `nul`
- `type`
- `string`
- `code`
- `closure`
- `int`
- `dir`
- `cmd`
- `function`
- `stream`
- `ipaddr`
- `macaddr`

See also: `type`

Returns A type value

Example

```
> typeof typeof
closure
> typeof []
dir
> typeof {}
code
> typeof ""
string
> typeof 0
int
> typeof NULL
nul
> typeof (typeof NULL!)
type
> eval str (typeof <>!)!
"dir"
set output [msg]: {
    if (equal (type "string"!) (typeof msg!)!) {
        echo msg!
    }{
        echo (str msg!)!
    }!
}
> output "this"
this
> output 34
34
```

tolower <string>

Name `tolower`

Kind Function

Arg syntax <chars:string>

Description Returns a string containing the (possibly multi-byte) characters in the argument string with any that have lower case equivalents replaced by it. The characters replaced may be encoded in more than one octet of the string. The number of octets in the string returned is not necessarily the same as those in the argument.
See also: `tolower`

Returns An upper-case string

Example
```
> echo ${toupper "ægis - ProtechT10N"!}
ÆGIS - PROTECHT10N
```

toupper <string>

Name `toupper`

Kind Function

Arg syntax <chars:string>

Description Returns a string containing the (possibly multi-byte) characters in the argument string with any that have upper case equivalents replaced by it. The characters replaced may be encoded in more than one octet of the string. The number of octets in the string returned is not necessarily the same as those in the argument.
See also: `tolower`

Returns An upper-case string

Example
```
> echo ${tolower "ÆGIS - ProTechT10N"!}
ægis - protecht10n
```

while <test> <do>

Name `while`

Kind Function

Arg syntax <test:code> <body:code>

Description while <test> evaluates non-zero execute <code>. Note that the test code is re-executed prior to each re-execution of the code and so it may refer to items that are updated by the code.

Returns Returns the value of the last <code> executed, or `NULL` otherwise.

Example

```
set which[app]: {
    .path = split ":" sys.env.PATH!;
    .fname = NULL;
    .n = 0;
    while { 0 == (cmp NULL fname!) and (inenv path n!) } {
        .fn = ""+(path.(n))+"/"+(app);
        .out = io.file fn "r"!;
        n = 1+(n);
        if 0 == (cmp NULL out!) {} {
            fname = fn; io.close out!;
        }!
    }!;
    fname
}
```

zip <dir> <dir>

Name chop

Kind Function

Arg syntax <dom:dir> <range:dir>

Description Normally takes two integer-indexed directories (vectors) and creates a directory with domain <dom> and range taken from values in <range>. The values for the range aree taken sequentially from values in <range>. If those values are exhausted <range> is repeated. When <range> is a single value (instead of a vector) it is used for every range value.
If the values in the <dom> vector are integers a new vector is created and if the values are strings a new named directory is created.
Currently values in <dom> that are not the same type as the first element in <dom> are ignored.
If there are no values in <dom> NULL is returned.

Returns A vector of successive substrings taken from the string.
See also: chop

Example

```
> zip <0..6> <"even", "odd">
<"even", "odd", "even", "odd", "even", "odd", "even">
> zip <3..0> <"abc", "def", "ghi", "jk">
<"jk", "ghi", "def", "abc">
> zip <0..9> 0
<0, 0, 0, 0, 0, 0, 0, 0, 0, 0>
> zip <"c","c#","d","d#","e","f","f#","g","g#","a","a#","b">
<0..11>
> [c=0, "c#"=1, d=2, "d#"=3, e=4, f=5, "f#"=6, g=7, "g#"=8,
a=9, "a#"=10, b=11]
set invert[x]:{zip (range x!) (domain x!)!}
> invert [c=0, "c#"=1, d=2, "d#"=3, e=4, f=5, "f#"=6, g=7,
"g#"=8, a=9, "a#"=10, b=11]
<"c", "c#", "d", "d#", "e", "f", "f#", "g", "g#", "a", "a#",
"b">
> invert [a=4,b=3,c=9,d=2]
<2="d", "b", "a", 9="c">
```

Implementation Notes

Directories

A number of types of directory are supported the most important of which are indexed and vector directories. A vector directory is implemented as a contiguous array with memory elements allocated for every index between a minimum and a maximum entry. Although some entries can be unset this type of directory will not store sparsely indexed data very well.
Vectors are generated when the '<' ... '>' syntax is used.

Name-indexed directories are generated when the '[' .. ']' syntax is used. Currently they can only be indexed by a string. This means that a vector (which is indexed by number) can not be used to hold name-indexed entries.

Garbage Collection

This implementation supports garbage collection of unreferenced values and also creates garbage quite freely. However it does not do garbage collection on-demand (e.g. when storage runs low) it will only perform garbage collection after the execution of a command line in command mode - and then only at the outer-most level of stack of `parser.exec` calls. This means that recursive or looping FTL expressions can currently generate garbage without limit.

Significance of Braces

When parsing a block of text between ‘{‘ and ‘}’ there is no escape mechanism to allow ‘}’ to be inserted into the text without a matching ‘{‘.