```
     main = "Bessel Functions  Y_nu(x)")
for(nu in nus){
    xx <- x[x > .6*nu]
    lines(xx, besselY(xx, nu=nu), col = nu+2)
}
legend(25, -.5, legend = paste("nu=", nus), col = nus+2, lwd = 1)

## negative nu in bessel_Y -- was bogus for a long time
curve(besselY(x, -0.1), 0, 10, ylim = c(-3,1), ylab = "")
for(nu in c(seq(-0.2, -2, by = -0.1)))
  curve(besselY(x, nu), add = TRUE)
title(expression(besselY(x, nu) * "   " *
                 {nu == list(-0.1, -0.2, ..., -2)}))
```

---

bindenv *Binding and Environment Locking, Active Bindings*

Description

These functions represent an interface for adjustments to environments and bindings within environments. They allow for locking environments as well as individual bindings, and for linking a variable to a function.

Usage

```
lockEnvironment(env, bindings = FALSE)
environmentIsLocked(env)
lockBinding(sym, env)
unlockBinding(sym, env)
bindingIsLocked(sym, env)

makeActiveBinding(sym, fun, env)
bindingIsActive(sym, env)
activeBindingFunction(sym, env)
```

Arguments

env	an environment.
bindings	logical specifying whether bindings should be locked.
sym	a name object or character string.
fun	a function taking zero or one arguments.

Details

The function `lockEnvironment` locks its environment argument. Locking the environment prevents adding or removing variable bindings from the environment. Changing the value of a variable is still possible unless the binding has been locked. The namespace environments of packages with namespaces are locked when loaded.

`lockBinding` locks individual bindings in the specified environment. The value of a locked binding cannot be changed. Locked bindings may be removed from an environment unless the environment is locked.

makeActiveBinding installs fun in environment env so that getting the value of sym calls fun with no arguments, and assigning to sym calls fun with one argument, the value to be assigned. This allows the implementation of things like C variables linked to R variables and variables linked to databases, and is used to implement setRefClass. It may also be useful for making thread-safe versions of some system globals. Currently active bindings are not preserved during package installation, but they can be created in .onLoad.

Value

The bindingIsLocked and environmentIsLocked return a length-one logical vector. The remaining functions return NULL, invisibly.

Author(s)

Luke Tierney

Examples

```
# locking environments
e <- new.env()
assign("x", 1, envir = e)
get("x", envir = e)
lockEnvironment(e)
get("x", envir = e)
assign("x", 2, envir = e)
try(assign("y", 2, envir = e)) # error

# locking bindings
e <- new.env()
assign("x", 1, envir = e)
get("x", envir = e)
lockBinding("x", e)
try(assign("x", 2, envir = e)) # error
unlockBinding("x", e)
assign("x", 2, envir = e)
get("x", envir = e)

# active bindings
f <- local( {
    x <- 1
    function(v) {
       if (missing(v))
           cat("get\n")
       else {
           cat("set\n")
           x <<- v
       }
       x
    }
})
makeActiveBinding("fred", f, .GlobalEnv)
bindingIsActive("fred", .GlobalEnv)
fred
fred <- 2
fred
```

`bitwise` *Bitwise Logical Operations*

Description

Logical operations on integer vectors with elements viewed as sets of bits.

Usage

```
bitwNot(a)
bitwAnd(a, b)
bitwOr(a, b)
bitwXor(a, b)

bitwShiftL(a, n)
bitwShiftR(a, n)
```

Arguments

`a, b`	integer vectors; numeric vectors are coerced to integer vectors.
`n`	non-negative integer vector of values up to 31.

Details

Each element of an integer vector has 32 bits.

Pairwise operations can result in integer `NA`.

Shifting is done assuming the values represent unsigned integers.

Value

An integer vector of length the longer of the arguments, or zero length if one is zero-length.

The output element is `NA` if an input is `NA` (after coercion) or an invalid shift.

See Also

The logical operators, `!`, `&`, `|`, `xor`. Notably these *do* work bitwise for `raw` arguments.

The classes "`octmode`" and "`hexmode`" whose implementation of the standard logical operators is based on these functions.

Package **bitops** has similar functions for numeric vectors which differ in the way they treat integers 2^{31} or larger.

Examples

```
bitwNot(0:12) # -1 -2  ... -13
bitwAnd(15L, 7L) #  7
bitwOr (15L, 7L) # 15
bitwXor(15L, 7L) #  8
bitwXor(-1L, 1L) # -2

## The "same" for 'raw' instead of integer :
```

```
rr12 <- as.raw(0:12) ; rbind(rr12, !rr12)
c(r15 <- as.raw(15), r7 <- as.raw(7)) #  0f 07
r15 & r7    # 07
r15 | r7    # 0f
xor(r15, r7)# 08

bitwShiftR(-1, 1:31) # shifts of 2^32-1 = 4294967295
```

body — *Access to and Manipulation of the Body of a Function*

Description

Get or set the *body* of a function which is basically all of the function definition but its formal arguments (`formals`), see the 'Details'.

Usage

```
body(fun = sys.function(sys.parent()))
body(fun, envir = environment(fun)) <- value
```

Arguments

`fun`	a function object, or see 'Details'.
`envir`	environment in which the function should be defined.
`value`	an object, usually a language object: see section 'Value'.

Details

For the first form, `fun` can be a character string naming the function to be manipulated, which is searched for from the parent frame. If it is not specified, the function calling body is used.

The bodies of all but the simplest are braced expressions, that is calls to `{`: see the 'Examples' section for how to create such a call.

Value

body returns the body of the function specified. This is normally a language object, most often a call to `{`, but it can also be a `symbol` such as `pi` or a constant (e.g., `3` or `"R"`) to be the return value of the function.

The replacement form sets the body of a function to the object on the right hand side, and (potentially) resets the `environment` of the function, and drops `attributes`. If `value` is of class `"expression"` the first element is used as the body: any additional elements are ignored, with a warning.

See Also

The three parts of a (non-primitive) function are its `formals`, body, and `environment`.

Further, see `alist`, `args`, `function`.

Examples

```
body(body)
f <- function(x) x^5
body(f) <- quote(5^x)
## or equivalently  body(f) <- expression(5^x)
f(3) # = 125
body(f)

## creating a multi-expression body
e <- expression(y <- x^2, return(y)) # or a list
body(f) <- as.call(c(as.name("{"), e))
f
f(8)

## Using substitute() may be simpler than 'as.call(c(as.name("{",..)))':
stopifnot(identical(body(f), substitute({ y <- x^2; return(y) })))
```

`bquote` *Partial substitution in expressions*

Description

An analogue of the LISP backquote macro. `bquote` quotes its argument except that terms wrapped in `.()` are evaluated in the specified `where` environment. If `splice = TRUE` then terms wrapped in `..()` are evaluated and spliced into a call.

Usage

```
bquote(expr, where = parent.frame(), splice = FALSE)
```

Arguments

`expr`	A language object.
`where`	An environment.
`splice`	Logical; if `TRUE` splicing is enabled.

Value

A language object.

See Also

`quote`, `substitute`

Examples

```
require(graphics)

a <- 2

bquote(a == a)
quote(a == a)
```

```
bquote(a == .(a))
substitute(a == A, list(A = a))

plot(1:10, a*(1:10), main = bquote(a == .(a)))

## to set a function default arg
default <- 1
bquote( function(x, y = .(default)) x+y )

exprs <- expression(x <- 1, y <- 2, x + y)
bquote(function() {..(exprs)}, splice = TRUE)
```

---

browser *Environment Browser*

Description

Interrupt the execution of an expression and allow the inspection of the environment where `browser` was called from.

Usage

```
browser(text = "", condition = NULL, expr = TRUE, skipCalls = 0L)
```

Arguments

`text`	a text string that can be retrieved once the browser is invoked.
`condition`	a condition that can be retrieved once the browser is invoked.
`expr`	a "condition". By default, and whenever not false after being coerced to `logical`, the debugger will be invoked, otherwise control is returned directly.
`skipCalls`	how many previous calls to skip when reporting the calling context.

Details

A call to `browser` can be included in the body of a function. When reached, this causes a pause in the execution of the current expression and allows access to the R interpreter.

The purpose of the `text` and `condition` arguments are to allow helper programs (e.g., external debuggers) to insert specific values here, so that the specific call to browser (perhaps its location in a source file) can be identified and special processing can be achieved. The values can be retrieved by calling `browserText` and `browserCondition`.

The purpose of the `expr` argument is to allow for the illusion of conditional debugging. It is an illusion, because execution is always paused at the call to browser, but control is only passed to the evaluator described below if `expr` is not `FALSE` after coercion to logical. In most cases it is going to be more efficient to use an `if` statement in the calling program, but in some cases using this argument will be simpler.

The `skipCalls` argument should be used when the `browser()` call is nested within another debugging function: it will look further up the call stack to report its location.

At the browser prompt the user can enter commands or R expressions, followed by a newline. The commands are `c` exit the browser and continue execution at the next statement.

`cont` synonym for `c`.

`f` finish execution of the current loop or function

`help` print this list of commands

`n` evaluate the next statement, stepping over function calls. For byte compiled functions interrupted by `browser` calls, `n` is equivalent to `c`.

`s` evaluate the next statement, stepping into function calls. Again, byte compiled functions make `s` equivalent to `c`.

`where` print a stack trace of all active function calls.

`r` invoke a `"resume"` restart if one is available; interpreted as an R expression otherwise. Typically `"resume"` restarts are established for continuing from user interrupts.

`Q` exit the browser and the current evaluation and return to the top-level prompt.

Leading and trailing whitespace is ignored, except for an empty line. Handling of empty lines depends on the `"browserNLdisabled"` option; if it is `TRUE`, empty lines are ignored. If not, an empty line is the same as `n` (or `s`, if it was used most recently).

Anything else entered at the browser prompt is interpreted as an R expression to be evaluated in the calling environment: in particular typing an object name will cause the object to be printed, and `ls()` lists the objects in the calling frame. (If you want to look at an object with a name such as `n`, print it explicitly, or use autoprint via `(n)`.

The number of lines printed for the deparsed call can be limited by setting `options(deparse.max.lines)`.

The browser prompt is of the form `Browse[`*n*`]>`: here `var{n}` indicates the 'browser level'. The browser can be called when browsing (and often is when `debug` is in use), and each recursive call increases the number. (The actual number is the number of 'contexts' on the context stack: this is usually `2` for the outer level of browsing and `1` when examining dumps in `debugger`.)

This is a primitive function but does argument matching in the standard way.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`debug`, and `traceback` for the stack on error. `browserText` for how to retrieve the text and condition.

`browserText` *Functions to Retrieve Values Supplied by Calls to the Browser*

Description

A call to browser can provide context by supplying either a text argument or a condition argument. These functions can be used to retrieve either of these arguments.

Usage

```
browserText(n = 1)
browserCondition(n = 1)
browserSetDebug(n = 1)
```

Arguments

`n`	The number of contexts to skip over, it must be non-negative.

Details

Each call to `browser` can supply either a text string or a condition. The functions `browserText` and `browserCondition` provide ways to retrieve those values. Since there can be multiple browser contexts active at any time we also support retrieving values from the different contexts. The innermost (most recently initiated) browser context is numbered 1: other contexts are numbered sequentially.

`browserSetDebug` provides a mechanism for initiating the browser in one of the calling functions. See `sys.frame` for a more complete discussion of the calling stack. To use `browserSetDebug` you select some calling function, determine how far back it is in the call stack and call `browserSetDebug` with `n` set to that value. Then, by typing `c` at the browser prompt you will cause evaluation to continue, and provided there are no intervening calls to browser or other interrupts, control will halt again once evaluation has returned to the closure specified. This is similar to the up functionality in gdb or the "step out" functionality in other debuggers.

Value

`browserText` returns the text, while `browserCondition` returns the condition from the specified browser context.

`browserSetDebug` returns NULL, invisibly.

Note

It may be of interest to allow for querying further up the set of browser contexts and this functionality may be added at a later date.

Author(s)

R. Gentleman

See Also

`browser`

`builtins` *Returns the Names of All Built-in Objects*

Description

Return the names of all the built-in objects. These are fetched directly from the symbol table of the R interpreter.