Package 'CodeDepends'

July 17, 2018

Version 0.6.5

Title Analysis of R Code for Reproducible Research and Code Comprehension

Description Tools for analyzing R expressions or blocks of code and determining the dependencies between them. It focuses on R scripts, but can be used on the bodies of functions. There are many facilities including the ability to summarize or get a high-level view of code, determining dependencies between variables, code improvement suggestions.

Author Duncan Temple Lang, Roger Peng, Deborah Nolan, Gabriel Becker

Maintainer Gabriel Becker <becker.gabriel@gene.com>

License GPL

URL https://github.com/duncantl/CodeDepends

BugReports https://github.com/duncantl/CodeDepends/issues

Depends methods

Imports codetools, graph, XML, utils

Suggests Rgraphviz, RUnit, knitr, highlight, RJSONIO, RCurl, Rcpp

VignetteBuilder knitr

Collate classes.R librarySymbols.R functionHandlers.R codeDepends.R sectionDepends.R sweave.R xml.R jss.R frags.R codeTypes.R gc.R graph.R parallel.R deps.R separateBlocks.R callGraph.R isPlot.R isOutput.R refScript.R sideEffects.R highlightCode.R freeVariables.R convenienceFuns.R

NeedsCompilation no

Repository CRAN

Date/Publication 2018-07-17 21:10:03 UTC

R topics documented:

`asVarName` *asVarName*

Description

This function grabs a symbol out of an expression and returns it as a character (see details for which symbol will be used).

This is a convenience function for use when constructing custom function handlers, it's unlikely to have much utility outside of that context.

Usage

```
asVarName(x)
```

Arguments

`x`	The (sub)expression to extract a symbol from

Details

This function always returns a character vector representing a single symbol from x, but which code varies depending on the exact form of x. When

- x is a single symbolthe character representation of the symbol is returned
- x is a function callasVarName is recursively called on the sub-expression for the first argument
- x is an assignmentasVarName is called recursively on the right-hand side (after -> expressions are transformed to <- ones). This is a special case of the rule above.

Value

A character vector of length one representing the symbol (or literal) as described in the Details section.

Author(s)

Duncan Temple Lang

Examples

```
asVarName(quote(rnorm(x, y, z))) # "x"
asVarName(quote(rnorm(x, y, z))[[1]] ) # "rnorm" b/c [[1]] is called fun
asVarName(quote(rownames(a) <- b )) # "a"
asVarName(quote(rnorm(10, y, z))) # "10"
```

`findWhenUnneeded` *Determine the code block after which a variable can be explicitly removed*

Description

These functions analyze the meta-information about code blocks and determine when a variable is no longer needed and can add code to the relevant code block to remove the variable.

Usage

```
findWhenUnneeded(var, frags, info = lapply(frags, getInputs), simplify,
                 index = TRUE, end = NA, redefined = FALSE)
addRemoveIntermediates(doc, frags = readScript(doc),
                       info = getInputs(frags),
                       vars = getVariables(info))
```

Arguments

var	the name of the variable(s) whose final
doc	the location of the script, given as a file name or a connection
frags	an object of class Script which is a list containing the code blocks in the script. This is typically obtained via a call to readScript.
info	an object of class ScriptInfo which is a list of ScriptNodeInfo objects.
simplify	ignored
index	a logical value indicating whether findWhenUnneeded should return the indices of the code blocks/fragments or the code fragments themselves.
vars	the names of all the variables of interest
end	the value to use if the variable is used in the last code block, i.e. the end of the script.
redefined	a logical value which controls whether we return the earliest code block in which the variable is redefined rather than when the variable is no longer used. Redefinition is a kind of "no longer being used" but for the value, not the variable.

Value

A vector of indices indicating the last expression in which each of the specified variables is an input.

Author(s)

Duncan Temple Lang

See Also

readScript addRemoveIntermediates

Examples

```
f = system.file("samples", "cleanVars.R", package = "CodeDepends")
sc = readScript(f)
findWhenUnneeded("x", sc)
findWhenUnneeded(c("x", "y"), sc)

# z is never used
findWhenUnneeded("z", sc)
findWhenUnneeded("z", sc, end = 1L)

code = addRemoveIntermediates(f)
 # Note that rm(x), rm(y) and rm(d) are added.
code[c(4, 5, 6)]
```

funchandlers *Specifying custom processing behavior, Function handlers and handler factories*

Description

Custom behavior when processing calls to certain functions is implemented by specifying *function handlers* for those functions. This can be used to alter CodeDepends' behavior when it sees these functions, or if desired, to ignore them entirely when processing the parent expression.

Function handlers should never be called directly by end users.

CodeDepends attempts to provide reasonable defaults, in the form of the `defaultFuncHandlers` list, which should be suitable for most users.

Arguments

`e`	The (sub)expression being processed. This will be a call to the function your handler is assigned to work on.
`collector`	The input collector in use. Represents state as the expression tree is walked.
`basedir`	The base directory when checking if a string literal is a file path
`input`	Are we in a part of the whole expression that specifies inputs
`formulaInputs`	Are symbols within formulas to be counted as inputs (`TRUE`) or non-standardly evaluated variables (`FALSE`)
`update`	Are we in a part of the expression that indicates a variable's value is being updated (i.e., complex right hand side)
`pipe`	Are we in a direct pipe call
`nseval`	Should any symbols that appear to be inputs be treated as nonstandardly-evaluated instead
`...`	unused

Details

Custom handling of functions and, rarely, some types of non functions (currently only inlined `NativeSymbol` objects) by the `getInputs` function is specified via function handlers, which are passed in a named list to `inputCollector` when creating a collector for use by `getInputs`.

Function handlers should only be used to construct an input collector (i.e., as an argument to `inputCollector`). They should not ever be called directly by end users.

When creating new function handlers, they should accept the arguments specified above (other than those to the factories). The first argument, e, will be an expression representing a call to the function the handler is specified for, and second `collector` will be the collector object. Handlers are expected to recursively process all aspects of the call expression to the extent desired. This will often be done by calling getInputs again on, e.g., some or all arguments passed into the function call.

Function handlers are also expected to respect the `pipe` and `nseval` arguments they receive.

getDependsThread *Compute which code blocks in a script are inputs to define a variable*

Description

This function is used to determine which code blocks in an R "script" that are needed to define a particular variable. This finds the smallest complete set of expressions or code blocks that must be evaluated in order to define the specified variable(s). It omits expressions that do not provide outputs that are not used as inputs to (indirectly) define the speciied variable.

Usage

```
getDependsThread(var, info, reverse = TRUE)
```

Arguments

var	the name of a variable in the script
info	a list of the meta-information for each of the code elements in the script.
reverse	a logical value that determines whether we reverse the indices of the expressions or leave them as end-to-first.

Value

An integer vector giving the indices of the script code blocks which are required to define var.

Author(s)

Duncan Temple Lang

See Also

getExpressionThread readScript getVariables

Examples

```
 sc = readScript(system.file("samples", "dual.R", package =
"CodeDepends"))
sci = getInputs(sc, formulaInputs = TRUE) ## script has formula with no data.frame
 idx = getDependsThread("fit", sci)
```

getDetailedTimelines *Compute and plot life cycle of variables in code*

Description

These functions allow one to get and visualize information about when variables are defined, redefined and used within and across blocks of code in a script or the body of a function.

Usage

```
getDetailedTimelines(doc, info = getInputs(doc, ...), vars =
getVariables(info, functions = functions), functions=TRUE, ...)
## S3 method for class 'DetailedVariableTimeline'
plot(x, var.srt = 0,
                                          var.mar = round(max(4,
.5*max(nchar(levels(x$var)))))), var.cex = 1, main = attr(x, "scriptName"),...)
```

Arguments

doc	the name of a file or a connection which identifies the code to be analyzed
info	meta-information extracted from the code identifying the inputs and outputs. See getInputs.
vars	the variables of interest
functions	What type of functions should be included in the timeline: NULL for none, TRUE for locally defined only, NA for unknown provenance functions, or FALSE for non-locally-defined functions. Defaults to TRUE.
x	the DetailedVariableTimeline object being plotted
var.srt	rotation of the labels for the vertical axis listing the variables
var.mar	the number of lines to leave for the vertical axis. The labels for this are variable names so one often needs more space or to change the size of the labels.
var.cex	character expansion factor for the variable labels on the vertical axis.
main	the title of the plot
...	Passed to down to getInputs for the default info value in getDetailedTimelines and to underlying plotting functions for plot.DetailedTimelines.

Value

getDetailedTimelines returns a data frame with four columns: step, used, defined, and var. Step represents steps within the timeline, with the same value indicating that the described event are occuring at the same time. used indicates whether var was used at that step, and defined indicates whether var was defined. Many rows will have FALSE for both as the variable is not used in that code block.

Author(s)

Duncan Temple Lang

See Also

getInputs

Examples

```
f = system.file("samples", "results-multi.R", package = "CodeDepends")
sc = readScript(f)
dtm = getDetailedTimelines(sc, getInputs(sc))
plot(dtm)
table(dtm$var)

 # A big/long function
info = getInputs(arima0)
dtm = getDetailedTimelines(info = info)
plot(dtm, var.cex = .7, mar = 4, srt = 30)
```

`getExpressionThread` *Find the sequence of expressions needed to get to a certain point in the code*

Description

What's the difference between this and getVariableInputs, getVariableDepends, getSectionDepends?

This does not currently attempt to get the minimal subset of expressions within the code block. In other words, if there are extraneous expressions within these blocks that are not actually necessary, these are evaluated. This is important for expressions with side effects, e.g. writing files or generating plots.

Usage

```
getExpressionThread(target, expressions, info = lapply(expressions,
getInputs, ...), ...)
```

Arguments

`target`	either the index of the expression of interest in `expressions` or the names of the variables.
`expressions`	the list of expressions
`info`	a list of objects giving information about the inputs to each top-level expression in expressions.
`...`	Passed to `getInputs` if `info` is not explicitly specified.

Value

A Script-class object containing the subset of the code chunks pertinent to the target variable(s).

Author(s)

Duncan Temple Lang

See Also

getDependsThread

Examples

```
e = readScript(system.file("samples", "dual.R", package = "CodeDepends"))
getExpressionThread("fit", e)

getExpressionThread("y", e)
getExpressionThread("x", e)

getExpressionThread("k", e)

 # With several
s = readScript(system.file("samples", "sitepairs.R", package = "CodeDepends"))
o = getExpressionThread("covs", s)
```

---

getInputs *Get input and output variables and literals from R expressions*

Description

This function is used to analyze an R expression and identify the input and output variables in the expressions and related packages that are loaded and files that are referenced.

This might be better called getCodeDepends. It is not to be confused with `getVariableInputs`.

Usage

```
getInputs (e, collector = inputCollector(), basedir = ".", reset =
  FALSE, formulaInputs = FALSE, ...)
```

Arguments

`e`	the expression whose code we are to process
`collector`	an object which collects the different elements of interest in the code.
`basedir`	the directory for the code relative to which we can resolve file names.
`...`	additional parameters for methods

reset	a logical value that controls whether we call the collector's reset method before starting to process the expressions in the script.
formulaInputs	Logical indicating whether symbols appearing in formulas should be treated as inputs to the expression. Defaults to FALSE.

Value

A ScriptInfo object containing information about the expression(s) in e.

Things tracked include:

files	the names of any strings used as arguments or literal values that correspond to file names.
strings	A vector of literal strings which appeared in e
libraries	the names of any libraries explicitly loaded within this code.
inputs	a character vector naming the variables that are used as inputs to the computations in this collection of expressions.
outputs	a character vector giving the names of the variables that are assigned values in this block of code, including assignments to elements of a variable, e.g. the variable x in the expression x[[1]] <- 10.
updates	character vector of variables which receive new values when evaluating the expression, but must already exist. Note this does not currently catch some situations, so checking if any symbols appear in both inputs and outputs is still prudent.
functions	a named logical vector, where the names are the names of the functions called and the values indicate whether the function is local (TRUE), from a package (FALSE) or unknown (NA). Note that this is not recursive.
removes	a vector of variables which were removed (via the rm function) in e
nsevalVars	A vector of variables which appear in appear in e, but which are non-standarly evaluated and thus are not typical inputs. Note this classification is determined by the functionhandlers in use by collector.

Note

Users should never call getInputs.langauge directly. It is listed here due to the vagaries of R CMD check documentation checks.

Author(s)

Duncan Temple Lang

See Also

parse

Examples

```
frags = parse(system.file("samples", "dual.R", package = "CodeDepends"))
# formula involves non-df variables
inputs = lapply(frags, getInputs, formulaInputs=TRUE)
inputs
sapply(inputs, slot, "outputs")

 # Specify the base directory in which to resolve the file names.
getInputs(frags[[5]], basedir = system.file("samples", package = "CodeDepends"))


f = system.file("samples", "namedAnnotatedScript.R", package = "CodeDepends")
sc = readScript(f, "labeled")
getInputs(sc)
getInputs(sc[[2]])
```

---

getPropagateChanges *Determine which expressions to update when a variable changes*

Description

This function allows us to determine which subsequent expressions in the document need to be evaluated when a variable is assigned a new value. This is the "opposite" of determining on which variables a given variable depends; this is for identifying which variables and expressions need to be updated when a variable changes. This is of use when propogating changes to dependent expressions.

Usage

```
getPropagateChanges(var, expressions, info = lapply(expressions,
                    getInputs), recursive = FALSE, index = FALSE, envir
                    = globalenv(), eval = !missing(envir), verbose =
                    FALSE)
```

Arguments

`var`	the name of the variable which has changed
`expressions`	the list of all expressions in the document
`info`	information extracted from the expressions about the inputs to each expressions. See `getInputs`.
`recursive`	a logical value that controls whether to work recursively on the expressions
`index`	a logical value which controls whether we return the indices of the expressions that would need to be evaluated based on the change to the variable `var`, or if `index` is `FALSE`, we return the expressions themselves.

envir	the environment in which to evaluate the expressions
eval	a logical value controlling whether we evaluate the expressions or just return them
verbose	a logical value that controls whether we output information about the expressions and their evaluation on the R console.

Value

This returns either the expressions or the indices of the expressions that need to be re-evaluated due to a change in var.

Note

The returned expression do NOT include the expression which defines the variable var. Only expressions *after* that are included.

Author(s)

Duncan Temple Lang

See Also

getExpressionThread getDependsThread

Examples

```
sc = readScript(system.file("samples", "formula.R", package = "CodeDepends"))
info = getInputs(sc)
getPropagateChanges("x", sc, info = info)
getPropagateChanges("y", sc, info = info)
```

---

getVariableDepends *Determine dependencies for code blocks*

Description

These functions provide ways to determine which code blocks must be evaluated before others based on input and output variables. getVariableDepends is used to determine the code blocks that need to be run in order to define particlar variables. getSectionDepends

Usage

```
getVariableDepends(vars, frags, info = lapply(frags, getInputs, ...),
checkLibraries = FALSE, asIndex = FALSE, functions = TRUE, ...)
getSectionDepends(sect, frags, info = lapply(frags, getInputs, ...), index =
FALSE, ...)
```

Arguments

vars	the names of the variables of interest
frags	the blocks or groups of expressions from the document
info	the information about the fragments that identify the inputs. This is typically computed as the default value for the parameter but can be provided explicitly when the caller has alreay computed this and passes it to different functions.
index	a logical value that controls whether we return the indices of the fragments of interest (TRUE) or return the fragments themselves (FALSE)
sect	the index of the section/fragment to be analyzed
checkLibraries	a logical value
asIndex	a logical value that controls whether we return the expressions/code blocks or their indices.
functions	passed to getVariables. What kind of functions should be counted as variables (TRUE is local functions only, the default)
...	passed to getInputs. Ignored if info is explicitly specified.

Value

getVariableDepends returns a Script-class object consisting of the subset of code blocks relevant to the specified variables.

If asIndex is TRUE, getVariableDepends returns the indices of the code blocks in the original script.

Author(s)

Duncan Temple Lang

See Also

getPropagateChanges getExpressionThread

Examples

```
e = readScript(system.file("samples", "dual.R", package = "CodeDepends"))
getVariableDepends("fit", e, formulaInputs = TRUE)
getVariableDepends("fit", e, formulaInputs = TRUE, asIndex = TRUE)

getVariableDepends("y", e, asIndex = TRUE)
getVariableDepends("y", e)
```

getVariables *Get the names of the variables used in code*

Description

These functions and methods allow one to get the names of the variables used within a script or block of code and from various derived types.

Usage

```
getVariables(x, inputs = FALSE, functions = TRUE, ...)
```

Arguments

x	the object with information about the variables
inputs	a logical indicating wether to include the input variables or just return the output variables, i.e. those on the left hand side of of an assignment. Defaults to FALSE
functions	Indicates what types of functions should be included. NULL Logical or NULL. Indicages what kind of functions should be counted as variables: local (TRUE, the default)(default) indicates none, TRUE indicates user-defined or unknown provenance functions, and FALSE indicates all functions. Ignored if inputs is FALSE.
...	Passed to getInputs when generating script information to comput on.

Value

A character vector, with possibly repeated values, giving the names of the variables. If an annotated script was used, the vector is named by the sections of the script.

Author(s)

Duncan Temple Lang

See Also

readScript getInputs

Examples

```
f = system.file("samples", "namedAnnotatedScript.R", package = "CodeDepends")
sc = readScript(f, "labeled")
getVariables(sc)

getVariables(sc[[3]])
```

guessTaskType *Guess the type of high-level task of a code block*

Description

This attempts to infer the type of the task being performed. There is a small set of known task types, listed in `system.file("Vocabulary", package = "CodeDepends")`.

Currently this uses simple rules. In the future, we might use a classifier.

Usage

```
guessTaskType(e, info = getInputs(e))
```

Arguments

`e`	the code block to be analyzed. This can be a call or an expression. Typically it is an element of a Script-class, i.e. a `ScriptNode-class` object
`info`	meta-information about the

Value

A character vector giving the different task identifiers.

Author(s)

Duncan Temple Lang

See Also

`readScript`

Examples

```
guessTaskType(quote(plot(x, y)))

e = expression({
        d = read.table("myData.txt")
        d$abc = d$a + log(d$b)
        d[ d$foo == 1, ] = sample(n)
    })
guessTaskType(e)
```

`highlightCode` *Display R code with highlighting of variables, links to functions and packages*

Description

This function leverages the `highlight` package to create an HTML display of R code. It connects all instances of a variable in the code so that a viewer can move the mouse over a variable and see all uses of it in the code.

The motivations for this is to help navigate a script and to allow us to connect the code to plots of, for example, the time-line or life-span of variables in a script.

Usage

```
highlightCode(obj, out = NULL, addFunctionLinks = TRUE, checkURLs= TRUE,
              inline = TRUE, h = htmlRenderer(addFunctionLinks,
              checkURLs), css = system.file("CSS", "highlight.css",
              package = "CodeDepends"), jsCode =
              system.file("JavaScript", "highlightSymbols.js", package =
              "CodeDepends"))
```

Arguments

`obj`	the name of a file containing R code or an R expression or function. Currently, this needs to be a file.
`out`	the name of a file to which the HTML document is written, or `NULL` or `NA` to just return the in-memory document.
`addFunctionLinks`	how to generate the links for function calls. This can be `NULL` to have no links for function calls, or a logical value indicating whether to have links or not, or a function. If this is a function, it is called with a vector of function names and should return a character vector with links for each of them.
`checkURLs`	When sorting through possible link targets, should we check for existing local files OR URLs. Defaults to `TRUE`, if `FALSE` only locally existing files are checked for.
`inline`	a logical value indicating whether to put the CSS and JavaScript code directly into the HTML document or just refer to them.
`h`	the renderer to create the HTML. See `highlight`
`css`	the URL or local file name for the CSS content
`jsCode`	the URL or local file name for the JavaScript code for the highlighting of the variables.

Details

This uses the highlight function to create the basic information for the code. We provide our own renderer to provide the links for function calls and packages and to specify markup for the symbols. Then we post-process the resulting HTML document to add our own CSS content and JavaScript code.

Value

An HTML document or the name of the file to which it was written if out is specified.

Author(s)

Duncan Temple Lang

Examples

```
f = system.file("samples", "sitepairs.R", package = "CodeDepends")
## url checking takes a while, too long for CRAN example
highlightCode(f, "foo.html", checkURLs=FALSE)
```

---

historyAsScript *Convert R interactive history to a Script object*

Description

This function is a means to capture the history of R commands interactively entered at the prompt in this session (or saved across sessions) as a Script-class object. One can then analyze the expressions to find relationships between variables and commands, which are irrelevant, ...

Usage

```
historyAsScript()
```

Value

An object of class Script-class.

Author(s)

Duncan Temple Lang

See Also

readScript history

`inputCollector` *Create customized input/output collector for use in getInputs*

Description

Create a custom input collector which will be used by getInputs to process to collect various aspects of the code as it is being processed. Custom collector functions can be specified which will be called when a particular function is called within the code.One major use for this is leveraging knowledge of specific functions' behavior to track side effects relevant to a particular use-case.

Usage

```
inputCollector(..., functionHandlers = list(...), inclPrevOutput =
FALSE, checkLibrarySymbols = FALSE, funcsAsInputs = checkLibrarySymbols)
```

Arguments

`...`	Custom information collection functions. Argument names correspond to R functions, with the custom collection function being called when a call to the named function is detected within the code being processed. Overridden by `functionHandlers`
`functionHandlers`	A named list of custom collection functions.
`inclPrevOutput`	Should variables which were output previously within the code be treated as inputs in subsequent expressions. If `TRUE` each expression within the code is treated separately with respect to detecting input variables, if `FALSE` the code is treated as a single block. Defaults to `FALSE`
`checkLibrarySymbols`	If `TRUE` symbols exported by default package and packages loaded within the code via `library` or `require` calls are tracked and excluded from the list of input variables. Defaults to`FALSE`
`funcsAsInputs`	If `TRUE` functions called by the code being processed are treated as input variables and listed as such. Defaults to the value of `checkLibrarySymbols`. A value of `funcsAsInputs` which does not agree with the value of `checkLibrarySymbols` is NOT recommended.

Details

Each custom collection function should accept three arguments:

e: the code or expression currently being processed

collector: the current inputCollector

basedir: the base directory in which the processing is taking place, e.g. to determine whether strings correspond to files

These functions should process the expression and then use `collector`'s collection functions and/or the <<- assignment operator to update the lists of found entities.

Currently trackable entities, updatable by <entity><<-c(<entity>, value) or as specified, include:

libraries: libraries loaded by the code via `library` or `require`. Updatable by calling `collector$library`

libSymbols: symbols exported by available libraries. Tracked automatically within `collector$library`

files: string constants which correspond to an existing file in `basedir`. Tracked automatically when strings are passed to `collector$string`

strings: string constants which do not correspond to existing files. Tracked automatically when strings are passed to `collector$string`

vars: all variable names used in the code. Updatable by calling `collector$vars` with `input` as `TRUE` or `FALSE` as appropriate

set: variable names which are assigned to in the code (input variables). Updatable by calling `collector$set` or `collector$vars` with `input=TRUE`

functions: functions called by the code. Updatable by calling `collector$calls`. This will also update `vars` if the collector was created with `funcsAsInputs=TRUE`

removes: variables removed by the vode via calls to `collector$removes`

updates: variables which have had elements within them updated, e.g. via `x$foo <- bar`. Updatable via calls to `collector$update`

sideEffects: side effects generated by the code. Experimental, default side effect detection should not be assumed to be robust or exhaustive. Updatable via calls to `sideEffects`

formulaVariables: If `formulaInputs` is `FALSE` within the call to getInputs, this tracks variables which appear within formulas, otherwise this is unused and such variables are treated as input. Updatable via the modelVars argument in calls to `collector$addInfo`

Value

A list of functions used to update internal tracking variables (see Details) as well as the following:

functionHandlers: The list of function handlers in use by the collector.

reset: A function which resets the internal tracking variables.

results: A function which returns a `ScriptNodeInfo` object representing the current state of the collection.

Note

Custom handlers take precedence over default proccessing mechanism. Care should be taken when overriding core functions such as `=`, `~`, `$`, `library`, etc.

Note

Specific internal behaviors of the default collection mechanisms are experimental and may change in future versions.

Author(s)

Duncan Temple Lang

See Also

ScriptNodeInfo getInputs

Examples

```
 f = system.file("samples", "results-multi.R", package="CodeDepends")
 sc = readScript(f)
collector = inputCollector(library = function(e, collector, basedir, ...)
{
  print(paste("loaded library",  e[[2]]))
  collector$library(as.character(e[[2]]))
})
res = getInputs(sc, collector = collector )
#[1] "loaded library splines"
#[1] "loaded library tsModel"
```

---

makeCallGraph *Create a graph representing which functions call other functions*

Description

This function and its methods provide facilities for constructing a graph representing which functions call which other functions.

Usage

```
makeCallGraph(obj, all = FALSE, ...)
```

Arguments

obj	The name of one or more packages as a string, optionally prefixed with "package:". This can be a vector of package names. Currently the packages should already be on the search path. Other inputs may be supported in the future
all	a logical value that controls whether the graph includes all the functions called by any of the target functions. This will greatly expand the graph.
...	additional parameters for the methods

Value

An object of class graphNEL-class

Note

We may extend this to deal with global variables and methods

Author(s)

Duncan Temple Lang

See Also

The graph and Rgraphviz packages.

The SVGAnnotation package can be used to mae thee graphs interactive.

Examples

```
gg = makeCallGraph("package:CodeDepends")
if(require(Rgraphviz)) {
   plot(gg, "twopi")

   ag = agopen(gg, layoutType = "circo", name = "bob")
   plot(ag)
}

if(require(Rgraphviz)) {
    # Bigger fonts.
  zz = layoutGraph(gg)
  graph.par(list(nodes = list(fontsize = 48)))
  renderGraph(zz)
}

  # Two packages
  library(codetools)
  gg = makeCallGraph(c("package:CodeDepends", "package:codetools"))
```

---

makeTaskGraph *Create a graph connecting the tasks within a script*

Description

This function create a graph connecting the high-level tasks within a script. The tasks are blocks of code that perform a step in the process. Each code block has input and output variables. These are used to define the associations between the tasks and which tasks are inputs to others and outputs that lead into others.

Usage

```
makeTaskGraph(doc, frags = readScript(doc), info = as(frags, "ScriptInfo"))
```

Arguments

doc	the name of the script file
frags	the code blocks in the script
info	the meta-information detailing the inputs and outputs of the different code blocks/fragments

Value

An object of class graphNEL-class.

Author(s)

Duncan Temple Lang

See Also

readScript getInputs

Examples

```
## Not run:
 f = system.file("samples", "dual.R", package = "CodeDepends")
 g = makeTaskGraph(f)

 if(require(Rgraphviz))
   plot(g)

 f = system.file("samples", "parallel.R", package = "CodeDepends")
 g = makeTaskGraph(f)

 if(require(Rgraphviz))
   plot(g)

 f = system.file("samples", "disjoint.R", package = "CodeDepends")
 g = makeTaskGraph(f)

 if(require(Rgraphviz))
   plot(g)

## End(Not run)
```

---

makeVariableGraph *Create a graph describing the relationships between variables in a script*

Description

This creates a graph of nodes and edges describing the relationship of how some variables are used in defining others.

Usage

```
makeVariableGraph(doc, frags = readScript(doc), info = getInputs(frags),
                  vars = getVariables(info, inputs = free), free = TRUE)
```

Arguments

doc	the name of the script file
frags	the code fragments from the script as a Script object.
info	the ScriptInfo list of ScriptNodeInfo objects describing each node.
vars	a character vector giving the names of the variables in the scripts. By default, these are the variables defined in the script.
free	a logical value that is passed to getInputs and controls whether we include the free/global variables in the script.

Details

Note that this collapses variables with the same name into a single node. Therefore, if the code uses the same name for two unrelated variables, there may be some confusion.

Value

An object of class graphNEL from the graph package.

Author(s)

Duncan Temple Lang

See Also

readScript getInputs getVariables

graph Rgraphviz

Examples

```
## Not run:
 u = url("http://www.omegahat.net/CodeDepends/formula.R")
 sc = readScript(u)
 close(u)
 g = makeVariableGraph(, sc)

## End(Not run)

 f = system.file("samples", "results-multi.R", package = "CodeDepends")
 sc = readScript(f)
 g = makeVariableGraph( info = getInputs(sc))
 if(require(Rgraphviz))
   plot(g)
```

`readScript` *Read the code blocks/chunks from a document*

Description

This is a general function that determines the type of the document and then extracts the code from it.

This is an S4 generic and so can be extended by other packages for document types that have a class, e.g. Word or OpenOffice documents.

`readAnnotatedScript` is for reading scripts that use a vocabulary to label code blocks with high-level task identifiers to indicate what the code does in descriptive terms.

Usage

```
readScript(doc, type = NA, txt = readLines(doc), ...)
readAnnotatedScript(doc, txt = readLines(doc))
```

Arguments

`doc`	the document, typically a string giving the file name. This can also be a connection, e.g. created via `url`.
`type`	a string indicating the type of the document. If this is missing, the function calls `getDocType` to attempt to determine this based on the "common" types of documents.
`txt`	the lines of text of the document.
`...`	Passed to low-level input functions used by various methods.

Value

A list of the R expressions that constitute the code blocks.

Author(s)

Duncan Temple Lang

See Also

`parse`

Examples

```
  e = readScript( system.file ("samples", "dual.R", package = "CodeDepends") )
  ## Not run:
  readScript(url("http://www.omegahat.net/CodeDepends/formula.R"))

## End(Not run)
```

runUpToSection *Evaluate the code blocks up to a particular section of a document*

Description

This function allows the caller to evaluate the code blocks within a document all the way up to a specified section of the document.

Usage

```
runUpToSection(section, doc, all = TRUE, env = globalenv(),
               nestedEnvironments = FALSE, frags = readScript(doc),
                verbose = FALSE)
```

Arguments

section	the index of the section, i.e. a number
doc	the name of the file containing the code
all	a logical value. It should be TRUE for now.
env	the environment in which the expressions will be evaluated
nestedEnvironments	a logical value controlling whether the each code block should be evaluated in its own environment which are created with the previous code block's environment as a parent environment.
frags	the code fragments read from the document or specified directly by the caller.
verbose	logical value indicating whether to display the code

Value

A list containing the results of evaluating the different fragments. The list will have a length given by the section number.

Note

Currently, `all = FALSE` is not implemented.

Author(s)

Duncan Temple Lang

See Also

sourceVariable

Examples

```
frags = parse(system.file("samples", "dual.R", package = "CodeDepends"))
runUpToSection(3, frags = frags, verbose = TRUE, all = TRUE)
```

Script-class *The Script class and elements*

Description

This package works with collections of expressions or code blocks and such a sequence can be thought of as a script. The Script class is a list of code elements. Such objects are typically created via a call to readScript. They can be read from XML files, tangled Sweave output, regular R source files and R source files that are annotated to identify the general task of each code block. This last type of script has its own class named AnnotatedScript and the code elements are annotated with labels such as dataInput, simulate, plot, model, eda and so on.

Each element of a Script list represents code. These are stored as objects of class ScriptNode. A ScriptNode class has slots for the code, the taskType indicating the high-level nature of the code, and an id so we can easily refer to it.

While our focus is on the code elements in a Script, we work with meta-data about the code elements. We identify information such as the input variables required by a code element, the variables it assigns (the outputs) and so on. This information is stored in a ScriptNodeInfo object. And a collection of such objects that parallels a script is a ScriptInfo object.

We can easily map a Script or a ScriptNode to the corresponding meta-information via the coercion methods as(script, "ScriptInfo") and as(node, "ScriptNodeInfo").

Objects from the Class

Objects of class Script are created with readScript.

Objects of class ScriptInfo are created with getInputs or as(, "ScriptInfo").

Slots

.Data: the elements of the list.

location: a character string that gives the file name or URL of the code for this script.

Extends

Class "list", from data part. Class "vector", by class "list", distance 2.

Methods

coerce signature(from = "Script", to = "ScriptInfo"): convert a Script to a ScriptInfo to access the meta-information

coerce signature(from = "ScriptNode", to = "ScriptNodeInfo"): compute the meta-information from an individual code element.

Author(s)

Duncan Temple Lang

See Also

readScript

Examples

```
f = system.file("samples", "results-multi.R", package = "CodeDepends")
sc = readScript(f)
info = as(sc, "ScriptInfo")
info = getInputs(sc, basedir = dirname(f))

  # Providing our own handler for calls to source()
sourceHandler = function(e, collector = NULL, basedir = ".", ...) {
    collector$string(e[[2]], , TRUE)
    collector$calls(as.character(e[[1]]))
}
h = CodeDepends:::inputCollector(source = sourceHandler)
info = getInputs(sc, h, basedir = dirname(f))

## Not run:
u = url("http://www.omegahat.net/CodeDepends/formula.R")
sc = readScript(u)
as(sc, "ScriptInfo")

## End(Not run)
```

---

separateExpressionBlocks

Convert a script into individual top-level calls

Description

This function converts a script of code blocks (e.g. from Sweave, XML, or an annotated script) with grouped expressions into individual top-level calls. The intent of this is to allow us to deal with the calls at a higher-level of granularity than code blocks. In other words, we can easily compute the depenendcies on the individual calls rather than on collections of calls. This allows us to re-evaluate individual expressions rather than entire code blocks when we have to update variables due to changes in "earlier" variables, i.e. those defined earlier in the script and recomputed for various reasons.

Usage

```
separateExpressionBlocks(blocks)
```

Arguments

blocks	a list of the expressions or calls, i.e. the code blocks, in the script.

Value

A list of call or assignment expressions.

Author(s)

Duncan Temple Lang

See Also

readScript

Examples

```
f = system.file("samples", "dual.R", package = "CodeDepends")
sc = readScript(f)
separateExpressionBlocks(sc)
```

---

sourceVariable *Evaluate code in document in order to define the specified variables*

Description

This function allows the caller to evaluate the code within the document (or list of code chunks directly) in order to define one or more variables and then terminate. This is similar to runUpToSection but is oriented towards variables rather than particular code blocks.

Usage

```
sourceVariable(vars, doc, frags = readScript(doc), eval = TRUE, env = globalenv(),
                nestedEnvironments = FALSE, verbose = TRUE,
                 checkLibraries = eval, force = FALSE, first = FALSE,
                  info = lapply(frags, getInputs))
```

Arguments

vars	the names of the variables which are of interest. This need not include intermediate variables, but instead is the vector of names of the variables that the caller wants defined ultimately.
doc	the document containing the code blocks
frags	the code fragments
eval	whether to evaluate the necessary code blocks or just return them.
env	the environment in which to evaluate the code blocks.
nestedEnvironments	a logical value indicating whether to evaluate each of the different code blocks within their own environment that is chained to the previous one.

verbose	a logical value indicating whether to print the expression being evaluated before it is actually evaluated.
checkLibraries	a logical value that controls whether we check for functions that are not currently available and if there are any whether we add calls to load libraries in getVariableDepends.
force	a logical value that controls whether we evaluate the expressions if they variables appear to exist.
first	a logical value. This is intended to allow running up to the first instance of the variable, not all of them.
info	the information about each expression. This is computed automatically, but the caller can specify it to avoid redundant computations.

Value

If eval is TRUE, a list of the results of evaluating the code blocks. Alternatively, if eval is FALSE, this returns the expressions constituting the code blocks. In this case, the function is the same as getVariableDepends

Note

We should add a nestedEnvironments parameter as in runUpToSection. In fact, consolidate the code so it can be shared.

Author(s)

Duncan Temple Lang

See Also

getVariableDepends

Examples

```
f = system.file("samples", "dual.R", package = "CodeDepends")
e = readScript(f)
getVariableDepends("k", frags = e)
sourceVariable("k", frags = e, verbose = TRUE)
```

---

splitRedefinitions *Divide a script into separate lists of code based on redefinition of a variable*

Description

The purpose of this function is to take a script consisting of individual calls or code blocks and to divide it into separate blocks in which a particular variable has only one definition. Within each block the variable is assigned a new value.

At present, the code is quite simple and separates code blocks that merely alter an existing variable's characteristics, e.g. setting the names, an individual variable. Ideally we want to separate very different uses of a symbol/variable name which are unrelated. We will add more sophisticated code to (heuristically) detect such different uses, e.g. explicit assignments to a variable.

Separating these code blocks can make it easier to treat the definitions separately and the different stages of the script.

Usage

```
splitRedefinitions(var, info)
```

Arguments

var	the name of the variable whose redefinition will identify the different code blocks
info	a list of ScriptNodeInfo-class objects identifying the input and output variables for each code block.

Value

A list with as many elements as there are (re)definitions of the variable each being a list of code blocks.

Author(s)

Duncan Temple Lang

See Also

readScript

Examples

```
 sc = readScript(system.file("samples", "redef.R", package =
  "CodeDepends"))
scinfo = getInputs(sc)
 groups = splitRedefinitions("x", scinfo)
```

updatingScript *Create a Script object that re-reads the original file as needed*

Description

This function reads the code in a particular document and creates a Script-class object to represent the code and allow us to do analysis on that code. Unlike readScript, this object continues to read any updates to the original code file when we use this Script object in computations. This allows us to modify the original source interactively and concurrently with our R session and still have the script remain up-to-date with that code.

Usage

```
updatingScript(doc, ...)
```

Arguments

doc	the name/location of the document containing the R code
...	any additional arguments, passed to readScript.

Details

This uses a reference class to update state across calls.

Value

an object of class DynScript

Author(s)

Duncan Temple Lang

See Also

readScript

Examples

```
cat("x = 1:10\ny = 3*x + 7 + rnorm(length(x))\n",  file = "foo.R")
sc = updatingScript("foo.R")

as(sc, "Script")

con = file("foo.R", "at")
cat("z = x + y", file = con)
close(con)

as(sc, "Script")
```

Index