

R documentation

of all in ‘../man’

March 30, 2009

R topics documented:

Script-class	1
findWhenUnneeded	3
getDependsThread	4
getDetailedTimelines	4
readScript	6
getExpressionThread	7
getInputs	7
getPropagateChanges	9
getVariableDepends	9
getVariables	10
guessTaskType	11
makeCallGraph	12
makeVariableGraph	13
makeTaskGraph	14
separateExpressionBlocks	15
sourceVariable	16
splitRedefinitions	17
Index	18

Script-class	<i>The Script class and elements</i>
--------------	--------------------------------------

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)
as(sc, "ScriptInfo")
```

```
u = url("http://penguin.biostat.jhsph.edu/cpkg/49c0/90223e7b16d72240a928f69bccd72a0a164c")
sc = readScript(u)
as(sc, "ScriptInfo")
```

findWhenUnneeded	<i>Determine the code block after which a variable can be explicitly removed</i>
------------------	--

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), va
```

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 <code>Script</code> which is a list containing the code blocks in the script. This is typically obtained via a call to <code>readScript</code> .
info	an object of class <code>ScriptInfo</code> which is a list of <code>ScriptNodeInfo</code> objects.
simplify	ignored
index	a logical value indicating whether <code>findWhenUnneeded</code> 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.

Author(s)

Duncan Temple Lang

See Also

`readScript` `addRemoveIntermediates`

Examples

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

code = addRemoveIntermediates(f)
# Note that rm(x, y) is added to the 5th code block
code[[5]]
```

getDependsThread	<i>Compute which code blocks in a script are inputs to define a variable</i>
------------------	--

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 specified variable.

Usage

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

Arguments

<code>var</code>	the name of one or more variables in the script
<code>info</code>	a list of the meta-information for each of the code elements in the script.
<code>reverse</code>	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"))
idx = getDependsThread("fit", as(sc, "ScriptInfo"))
```

getDetailedTimelines	<i>Compute and plot life cycle of variables in code</i>
----------------------	---

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))
## S3 method for class 'DetailedVariableTimeline':
plot(x, var.srt = 0, var.mar = 5, var.cex = 1, ...)
```

Arguments

<code>doc</code>	the name of a file or a connection which identifies the code to be analyzed
<code>info</code>	meta-information extracted from the code identifying the inputs and outputs. See getInputs .
<code>vars</code>	the variables of interest
<code>x</code>	the <code>DetailedVariableTimeline</code> object being plotted
<code>var.srt</code>	rotation of the labels for the vertical axis listing the variables
<code>var.mar</code>	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.
<code>var.cex</code>	character expansion factor for the variable labels on the vertical axis.
<code>...</code>	additional arguments to the <code>plot</code> command. These might include, for example, <code>main</code> to put a title on the plot.

Value

`getDetailedTimelines` returns a data frame with three variables: `var`, `used` and `defined`. For each variable, there are as many rows as there are code blocks in the document (and elements in `info`). (Variables that are redefined will have more rows, but these are essentially different variables.) These rows correspond to the different code blocks or "time steps". `used` and `defined` indicate whether the variable acted as an input or was defined within this code block. Many will have `FALSE` for both as the variable is not used in that code block. `var` is used merely to identify the variable.

Author(s)

Duncan Temple Lang

See Also

[getInputs](#)

Examples

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

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

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

<code>doc</code>	the document, typically a string giving the file name. This can also be a connection, e.g. created via url .
<code>type</code>	a string indicating the type of the document. If this is missing, the function calls <code>getDocType</code> to attempt to determine this based on the "common" types of documents.
<code>txt</code>	the lines of text of the document.

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") )

readScript(url("http://penguin.biostat.jhsph.edu/cpkg/49c0/90223e7b16d72240a928f69bccd7
```

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
expressions
info
```

Author(s)

Duncan Temple Lang

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 = ".", ...)
```

Arguments

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

Value

A list with elements:

<code>files</code>	the names of any strings used as arguments or literal values that correspond to file names.
<code>libraries</code>	the names of any libraries explicitly loaded within this code.
<code>inputs</code>	a character vector naming the variables that are used as inputs to the computations in this collection of expressions.
<code>outputs</code>	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 <code>x</code> in the expression <code>x[[1]] <- 10</code> .
<code>functions</code>	a character vector naming the functions that are called within the code for this expression. This is not recursive, i.e. does not find the functions called by the function calls in this section.

Author(s)

Duncan Temple Lang

See Also

[parse](#)

Examples

```
frags = parse(system.file("samples", "dual.R", package = "CodeDepends"))
inputs = lapply(frag, getInputs)
inputs
sapply(inputs, slot, "outputs")

# Specify the base directory in which to resolve the file names.
getInputs(frag[[1]], 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 propagating changes to dependent expressions.

Usage

```
getPropagateChanges(var, expressions, info = lapply(expressions, getInputs), recursive = TRUE, index = 0)
```

Arguments

`var` the name of the variable which has changed
`expressions`
`info`
`recursive`
`index`

Author(s)

Duncan Temple Lang

See Also

[getExpressionThread](#) [getDependsThread](#)

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 particular variables. `getSectionDepends`

Usage

```
getVariableDepends(vars, frags, info = lapply(frags, getInputs))
getSectionDepends(sect, frags, info = lapply(frags, getInputs), index = FALSE)
```

Arguments

vars
frags
info
index
sect

Author(s)

Duncan Temple Lang

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, ...)
```

Arguments

x the object with information about the variables
... any additional parameters for methods

Value

A character vector, with possibly repeated values, giving the names of the variables.

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	<i>Guess the type of high-level task of a code block</i>
---------------	--

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

<code>e</code>	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 <code>ScriptNode-class</code> object
<code>info</code>	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)
```

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

<code>obj</code>	a function, the name of a function, the name of a package, a character vector of function names,
<code>all</code>	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.
<code>...</code>	additional parameters for the methods

Value

An object of class `graphNEL`

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 make the graphs interactive.

Examples

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

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

makeVariableGraph	<i>Create a graph describing the relationships between variables in a script</i>
-------------------	--

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 =
```

Arguments

```
doc
frags
info
vars
```

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

```
u = url("http://penguin.biostat.jhsph.edu/cpkg/49c0/90223e7b16d72240a928f69bccd72a0a164c")
sc = readScript(u)
library(Rgraphviz)
g = makeVariableGraph(, sc)

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

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 = getInputs(frags))
```

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

References

put references to the literature/web site here

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)
```

`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 dependencies 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	<i>Evaluate code in document in order to define the specified variables</i>
----------------	---

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 = FALSE)
```

Arguments

<code>vars</code>	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.
<code>doc</code>	the document containing the code blocks
<code>frags</code>	the code fragments
<code>eval</code>	whether to evaluate the necessary code blocks or just return them.
<code>env</code>	the environment in which to evaluate the code blocks.
<code>nestedEnvironments</code>	a logical value indicating whether to evaluate each of the different code blocks within their own environment that is chained to the previous one.
<code>verbose</code>	a logical value indicating whether to print the expression being evaluated before it is actually evaluated.

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 bloc 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

<code>var</code>	the name of the variable whose redefinition will identify the different code blocks
<code>info</code>	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", "dual.R", package = "CodeDepends"))
groups = separateExpressionBlocks(sc)
```

Index

*Topic IO

readScript, 5
separateExpressionBlocks, 14

*Topic classes

Script-class, 1

*Topic hplot

getDetailedTimelines, 4
makeCallGraph, 11
makeVariableGraph, 12

*Topic programming

findWhenUnneeded, 2
getDependsThread, 3
getDetailedTimelines, 4
getExpressionThread, 6
getInputs, 7
getPropagateChanges, 8
getVariableDepends, 9
getVariables, 9
guessTaskType, 10
makeCallGraph, 11
makeTaskGraph, 13
makeVariableGraph, 12
readScript, 5
Script-class, 1
separateExpressionBlocks, 14
sourceVariable, 15
splitRedefinitions, 16

addRemoveIntermediates, 3
addRemoveIntermediates
(*findWhenUnneeded*), 2
AnnotatedScript-class
(*Script-class*), 1

coerce, expression, ScriptNodeInfo-method
(*Script-class*), 1
coerce, language, ScriptNodeInfo-method
(*Script-class*), 1
coerce, Script, ScriptInfo-method
(*Script-class*), 1
coerce, ScriptNode, ScriptNodeInfo-method
(*Script-class*), 1

findWhenUnneeded, 2

getDependsThread, 3, 8
getDetailedTimelines, 4
getExpressionThread, 4, 6, 8
getInputs, 1, 4, 5, 7, 10, 12, 13
getInputs, ANY-method (*getInputs*),
7
getInputs, function-method
(*getInputs*), 7
getInputs, Script-method
(*getInputs*), 7
getInputs, ScriptNode-method
(*getInputs*), 7
getPropagateChanges, 8
getSectionDepends
(*getVariableDepends*), 9
getVariableDepends, 9, 15, 16
getVariables, 4, 9, 12
getVariables, Script-method
(*getVariables*), 9
getVariables, ScriptInfo-method
(*getVariables*), 9
getVariables, ScriptNode-method
(*getVariables*), 9
getVariables, ScriptNodeInfo-method
(*getVariables*), 9
graphNEL, 11
graphNEL-class, 13
guessTaskType, 10
list, 2
makeCallGraph, 11
makeCallGraph, character-method
(*makeCallGraph*), 11
makeCallGraph, list-method
(*makeCallGraph*), 11
makeTaskGraph, 13
makeVariableGraph, 12
parse, 6, 7
plot.DetailedVariableTimeline
(*getDetailedTimelines*), 4
readAnnotatedScript (*readScript*),
5

`readScript`, 1–4, 5, 10, 12, 13, 15, 17
`readScript`, character-method
 (`readScript`), 5
`readScript`, connection-method
 (`readScript`), 5

`Script`-class, 1, 10
`ScriptInfo`-class (`Script`-class), 1
`ScriptNode`-class (`Script`-class), 1
`ScriptNodeInfo`-class, 16
`ScriptNodeInfo`-class
 (`Script`-class), 1
`separateExpressionBlocks`, 14
`sourceVariable`, 15
`splitRedefinitions`, 16

`url`, 5

`vector`, 2