AnnotationDbi

April 19, 2009

AnnDbBimap-envirAPI

Environment-like API for AnnDbBimap objects

Description

These methods allow the user to manipulate any AnnDbBimap object as if it was an environment. This environment-like API is provided for backward compatibility with the traditional environment-based maps.

Usage

```
ls(name, pos, envir, all.names, pattern)
exists(x, where, envir, frame, mode, inherits)
get(x, pos, envir, mode, inherits)
#x[[i]]
#x$name

## Converting to a list
mget(x, envir, mode, ifnotfound, inherits)
eapply(env, FUN, ..., all.names)
#contents(object, all.names)

## Additional convenience method
sample(x, size, replace=FALSE, prob=NULL)
```

Arguments

name	An AnnDbBimap object for ls. A key as a literal character string or a name (possibly backtick quoted) for x$name.
pos	Ignored.
envir	Ignored for ls. An AnnDbBimap object for mget, get and exists.
all.names	Ignored.
pattern	An optional regular expression. Only keys matching 'pattern' are returned.
x	The key(s) to search for for exists, get and mget. An AnnDbBimap object for [[and x$name. An AnnDbBimap object or an environment for sample.

1

where	Ignored.
frame	Ignored.
mode	Ignored.
inherits	Ignored.
i	Single key specifying the map element to extract.
ifnotfound	A value to be used if the key is not found. Only NA is currently supported.
env	An [AnnDbBimap](#) object.
FUN	The function to be applied (see original [eapply](#) for environments for the details).
...	Optional arguments to FUN.
size	Non-negative integer giving the number of map elements to choose.
replace	Should sampling be with replacement?
prob	A vector of probability weights for obtaining the elements of the map being sampled.

See Also

[ls](#), [exists](#), [get](#), [mget](#), [eapply](#), [contents](#), [sample](#), [BimapFormatting](#), [Bimap](#)

Examples

```
library(hgu95av2.db)
x <- hgu95av2CHRLOC

ls(x)[1:3]
exists(ls(x)[1], x)
exists("titi", x)
get(ls(x)[1], x)
x[[ls(x)[1]]]
x$titi # NULL

mget(ls(x)[1:3], x)
eapply(x, length)
contents(x)

sample(x, 3)
```

AnnDbObj-objects   *AnnDbObj objects*

Description

The AnnDbObj class is the most general container for storing any kind of SQLite-based annotation data.

Details

Many classes in AnnotationDbi inherit directly or indirectly from the AnnDbObj class. One important particular case is the [AnnDbBimap](#) class which is the lowest class in the AnnDbObj hierarchy to also inherit the [Bimap](#) interface.

Accessor-like methods

In the code snippets below, `x` is an AnnDbObj object.

`dbconn(x)`: Return a connection object to the SQLite DB containing `x`'s data.

`dbfile(x)`: Return the path (character string) to the SQLite DB (file) containing `x`'s data.

`dbmeta(x, name)`: Print the value of metadata whose name is 'name'. Also works if `x` is a DBIConnection object.

`dbschema(x, file="", show.indices=FALSE)`: Print the schema definition of the SQLite DB. Also works if `x` is a DBIConnection object.

The `file` argument must be a connection, or a character string naming the file to print to (see the `file` argument of the [cat](#) function for the details).

The CREATE INDEX statements are not shown by default. Use `show.indices=TRUE` to get them.

`dbInfo(x)`: Prints other information about the SQLite DB. Also works if `x` is a DBIConnection object.

See Also

[dbConnect](#), [dbListTables](#), [dbListFields](#), [dbGetQuery](#), [Bimap](#)

Examples

```
library("hgu95av2.db")

dbconn(hgu95av2ENTREZID)              # same as hgu95av2_dbconn()
dbfile(hgu95av2ENTREZID)              # same as hgu95av2_dbfile()

dbmeta(hgu95av2_dbconn(), "ORGANISM")
dbmeta(hgu95av2_dbconn(), "DBSCHEMA")
dbmeta(hgu95av2_dbconn(), "DBSCHEMAVERSION")

library("DBI")
dbListTables(hgu95av2_dbconn())       #lists all tables on connection

## If you use dbSendQuery instead of dbGetQuery
## (NOTE: for ease of use, this is defintitely NOT reccomended)
## Then you may need to know how to list results objects
dbListResults(hgu95av2_dbconn())      #for listing results objects

## Sometimes you may want to see all the SQLite databases that are
## presently connected in your session. To do that you have to specify
## the driver:
library("RSQLite")
drvr <- dbDriver("SQLite")        #gets the driver for SQLite
dbListConnections(drvr)           #List all DB Connections using drvr

dbListFields(hgu95av2_dbconn(), "probes")
dbListFields(hgu95av2_dbconn(), "genes")
dbschema(hgu95av2ENTREZID)        # same as hgu95av2_dbschema()
## According to the schema, the probes._id column references the genes._id
## column. Note that in all tables, the "_id" column is an internal id with
## no biological meaning (provided for allowing efficient joins between
```

```
## tables).
## To retrieve the mapping between manufacturer IDs and Entrez Gene IDs:
dbGetQuery(hgu95av2_dbconn(), "SELECT * FROM probes INNER JOIN genes USING(_id) LIMIT 1
## This mapping is in fact the ENTREZID map:
toTable(hgu95av2ENTREZID)[1:10, ] # only relevant columns are retrieved

dbInfo(hgu95av2GO)                  # same as hgu95av2_dbInfo()

##Advanced example:
##Sometimes you may wish to join data from across multiple databases at
##once:
## In the following example we will attach the GO database to the
## hgu95av2 database, and then grab information from separate tables
## in each database that meet a common criteria.
library(hgu95av2.db)
library("GO.db")
attachSql <- paste('ATTACH "', GO_dbfile(), '" as go;', sep = "")
dbGetQuery(hgu95av2_dbconn(), attachSql)
sql <- 'SELECT  DISTINCT a.go_id AS "hgu95av2.go_id",
         a._id AS "hgu95av2._id",
         g.go_id AS "GO.go_id", g._id AS "GO._id",
         g.term, g.ontology, g.definition
         FROM go_bp_all AS a, go.go_term AS g
         WHERE a.go_id = g.go_id LIMIT 10;'
data <- dbGetQuery(hgu95av2_dbconn(), sql)
data
## For illustration purposes, the internal id "_id" and the "go_id"
## from both tables is included in the output.  This makes it clear
## that the "go_ids" can be used to join these tables but the internal
## ids can NOT.  The internal IDs (which are always shown as _id) are
## suitable for joins within a single database, but cannot be used
## across databases.
```

AnnDbPkg-checker   *Check the SQL data contained in an SQLite-based annotation package*

Description

Check the SQL data contained in an SQLite-based annotation package.

Usage

```
checkMAPCOUNTS(pkgname)
```

Arguments

pkgname   The name of the SQLite-based annotation package to check.

Author(s)

H. Pages

See Also

AnnDbPkg-maker

Examples

```
checkMAPCOUNTS("hgu95av2.db")
checkMAPCOUNTS("GO.db")
```

AnnDbPkg-maker *Creates an SQLite-based annotation package*

Description

Creates an SQLite-based annotation package from an SQLite file.

Usage

```
makeAnnDbPkg(x, dbfile, dest_dir=".", no.man=FALSE, ...)
loadAnnDbPkgIndex(file)
```

Arguments

x	A AnnDbPkgSeed object, a list, a string or a regular expression.
dbfile	The path to the SQLite containing the annotation data for the package to build.
dest_dir	The directory where the package will be created.
file	The path to a DCF file containing the list of annotation packages to build.
no.man	If TRUE then no man page is included in the package.
...	Extra args used for extra filtering.

See Also

[AnnDbPkg-checker](#)

Examples

```
## With a "AnnDbPkgSeed" object:
seed <- new("AnnDbPkgSeed",
    Package="hgu133a2.db",
    Version="0.0.99",
    PkgTemplate="HUMANCHIP.DB",
    AnnObjPrefix="hgu133a2"
)
if (FALSE)
    makeAnnDbPkg(seed, "path/to/hgu133a2.sqlite")

## With package names:
## (Note that in this case makeAnnDbPkg() will use the package descriptions
## found in the master index file ANNDBPKG-INDEX.TXT located in the
## AnnotationDbi package.)
if (FALSE)
    makeAnnDbPkg(c("hgu95av2.db", "hgu133a2.db"))

## A character vector of length 1 is treated as a regular expression:
if (FALSE)
    makeAnnDbPkg("hgu.*")
```

```
## To make all the packages described in the master index:
if (FALSE)
    makeAnnDbPkg("")
## Extra args can be used to narrow down the roaster of packages to make:
if (FALSE) {
    makeAnnDbPkg("", PkgTemplate="HUMANCHIP.DB", manufacturer="Affymetrix")
    makeAnnDbPkg(".*[3k]\\.db", species=c("Mouse", "Rat"))
}

## The master index file ANNDBPKG-INDEX.TXT can be loaded with:
loadAnnDbPkgIndex()
```

Bimap-direction   *Methods for getting/setting the direction of a Bimap object, and undi-*
                         rected methods for getting/counting/setting its keys

---

Description

These methods are part of the Bimap interface (see ?Bimap for a quick overview of the Bimap objects and their interface).

They are divided in 2 groups: (1) methods for getting or setting the direction of a Bimap object and (2) methods for getting, counting or setting the left or right keys (or mapped keys only) of a Bimap object. Note that all the methods in group (2) are undirected methods i.e. what they return does NOT depend on the direction of the map (more on this below).

Usage

```
## Getting or setting the direction of a Bimap object
direction(x)
direction(x) <- value
revmap(x, ...)

## Getting, counting or setting the left or right keys (or mapped
## keys only) of a Bimap object
Lkeys(x)
Rkeys(x)
Llength(x)
Rlength(x)
mappedLkeys(x)
mappedRkeys(x)
count.mappedLkeys(x)
count.mappedRkeys(x)
Lkeys(x) <- value
Rkeys(x) <- value
subset(x, ...)
```

Arguments

x	A Bimap object.
value	A single integer or character string indicating the new direction in direction(x) <- value. A character vector containing the new keys (must be a subset of the current keys) in Lkeys(x) <- value and Rkeys(x) <- value.

...   Extra argument for `revmap` can be:

          objName The name to give to the reversed map (only supported if `x` is an AnnDbBimap object).

          Extra arguments for `subset` can be:

          Lkeys The new Lkeys.

          Rkeys The new Rkeys.

          drop.invalid.keys If `drop.invalid.keys=FALSE` (the default), an error will be raised if the new Lkeys or Rkeys contain invalid keys i.e. keys that don't belong to the current Lkeys or Rkeys. If `drop.invalid.keys=TRUE`, invalid keys are silently dropped.

          objName The name to give to the submap (only supported if `x` is an AnnDb-Bimap object).

Details

All Bimap objects have a direction which can be left-to-right (i.e. the mapping goes from the left keys to the right keys) or right-to-left (i.e. the mapping goes from the right keys to the left keys). A Bimap object `x` that maps from left to right is considered to be a direct map. Otherwise it is considered to be an indirect map (when it maps from right to left).

`direction` returns 1 on a direct map and −1 otherwise.

The direction of `x` can be changed with `direction(x) <- value` where value must be 1 or −1. An easy way to reverse a map (i.e. to change its direction) is to do `direction(x) <- −direction(x)`, or, even better, to use `revmap(x)` which is actually the recommended way for doing it.

The `Lkeys` and `Rkeys` methods return respectively the left and right keys of a Bimap object. Unlike the `keys` method (see `?keys` for more information), these methods are direction-independent i.e. what they return does NOT depend on the direction of the map. Such methods are also said to be "undirected methods" and methods like the `keys` method are said to be "directed methods".

All the methods described below are also "undirected methods".

`Llength(x)` and `Rlength(x)` are equivalent to (but more efficient than) `length(Lkeys(x))` and `length(Rkeys(x))`, respectively.

The `mappedLkeys` (or `mappedRkeys`) method returns the left keys (or right keys) that are mapped to at least one right key (or one left key).

`count.mappedLkeys(x)` and `count.mappedRkeys(x)` are equivalent to (but more efficient than) `length(mappedLkeys(x))` and `length(mappedRkeys(x))`, respectively.

`Lkeys(x) <- value` and `Rkeys(x) <- value` are the undirected versions of `keys(x) <- value` (see `?keys` for more information) and `subset(x, Lkeys=new_Lkeys, Rkeys=new_Rkeys)` is provided as a convenient way to reduce the sets of left and right keys in one single function call.

Value

`1L` or `−1L` for `direction`.

A Bimap object of the same subtype as `x` for `revmap` and `subset`.

A character vector for `Lkeys`, `Rkeys`, `mappedLkeys` and `mappedRkeys`.

A single non-negative integer for `Llength`, `Rlength`, `count.mappedLkeys` and `count.mappedRkeys`.

Author(s)

H. Pages

See Also

Bimap, Bimap-keys, BimapFormatting, AnnDbBimap-envirAPI

Examples

```
library(hgu95av2.db)
ls(2)
x <- hgu95av2GO
x
summary(x)
direction(x)

length(x)
Llength(x)
Rlength(x)

keys(x)[1:4]
Lkeys(x)[1:4]
Rkeys(x)[1:4]

count.mappedkeys(x)
count.mappedLkeys(x)
count.mappedRkeys(x)

mappedkeys(x)[1:4]
mappedLkeys(x)[1:4]
mappedRkeys(x)[1:4]

y <- revmap(x)
y
summary(y)
direction(y)

length(y)
Llength(y)
Rlength(y)

keys(y)[1:4]
Lkeys(y)[1:4]
Rkeys(y)[1:4]

## etc...

## Get rid of all unmapped keys (left and right)
z <- subset(y, Lkeys=mappedLkeys(y), Rkeys=mappedRkeys(y))
```

Bimap-keys   *Methods for manipulating the keys of a Bimap object*

Description

These methods are part of the Bimap interface (see `?Bimap` for a quick overview of the Bimap objects and their interface).

Usage

```
keys(x)
#length(x)
isNA(x)
mappedkeys(x)
count.mappedkeys(x)
keys(x) <- value
#x[i]
```

Arguments

x	A Bimap object.
value	A character vector containing the new keys (must be a subset of the current keys).
i	A character vector containing the keys of the map elements to extract.

Details

keys(x) returns the set of all valid keys for map x. For example, keys(hgu95av2GO) is the set of all probe set IDs for chip hgu95av2 from Affymetrix. Note that the double bracket operator [[for Bimap objects is guaranteed to work only with a valid key and will raise an error if the key is invalid. (See ¿AnnDbBimap-envirAPI` for more information about this operator.)

length(x) is equivalent to (but more efficient than) length(keys(x)).

A valid key is not necessarily mapped ([[will return an NA on an unmapped key).

isNA(x) returns a logical vector of the same length as x where the TRUE value is used to mark keys that are NOT mapped and the FALSE value to mark keys that ARE mapped.

mappedkeys(x) returns the subset of keys(x) where only mapped keys were kept.

count.mappedkeys(x) is equivalent to (but more efficient than) length(mappedkeys(x)).

Two (almost) equivalent forms of subsetting a Bimap object are provided: (1) by setting the keys explicitly and (2) by using the single bracket operator [for Bimap objects. Let's say the user wants to restrict the mapping to the subset of valid keys stored in character vector mykeys. This can be done either with keys(x) <- mykeys (form (1)) or with y <- x[mykeys] (form (2)). Please note that form (1) alters object x in an irreversible way (the original keys are lost) so form (2) should be preferred.

All the methods described on this pages are "directed methods" i.e. what they return DOES depend on the direction of the Bimap object that they are applied to (see ?direction for more information about this).

Value

A character vector for keys and mappedkeys.

A single non-negative integer for length and count.mappedkeys.

A logical vector for isNA.

A Bimap object of the same subtype as x for x[i].

Author(s)

H. Pages

See Also

Bimap, AnnDbBimap-envirAPI, Bimap-toTable, BimapFormatting

Examples

```
library(hgu95av2.db)
x <- hgu95av2GO
x
length(x)
count.mappedkeys(x)
x[1:3]
links(x[1:3])

## Keep only the mapped keys
keys(x) <- mappedkeys(x)
length(x)
count.mappedkeys(x)
x # now it is a submap

## The above subsetting can also be achieved with
x <- hgu95av2GO[mappedkeys(hgu95av2GO)]

## mappedkeys() and count.mappedkeys() also work with an environment
## or a list
z <- list(k1=NA, k2=letters[1:4], k3="x")
mappedkeys(z)
count.mappedkeys(z)
```

Bimap-toTable   *Methods for manipulating a Bimap object in a data-frame style*

Description

These methods are part of the Bimap interface (see ?Bimap for a quick overview of the Bimap objects and their interface).

Usage

```
## Extract all the columns of the map (links + right attributes)
toTable(x)
nrow(x)
ncol(x)
#dim(x)
head(x, ...)
tail(x, ...)

## Extract only the links of the map
links(x)
count.links(x)
nhit(x)

## Col names and col metanames
```

```
colnames(x, do.NULL=TRUE, prefix="col")
colmetanames(x)
Lkeyname(x)
Rkeyname(x)
keyname(x)
tagname(x)
Rattribnames(x)
Rattribnames(x) <- value
```

Arguments

x	A [Bimap](#) object (or a list or an environment for `nhit`).
...	Further arguments to be passed to or from other methods (see [head](#) or [tail](#) for the details).
do.NULL	Ignored.
prefix	Ignored.
value	A character vector containing the names of the new right attributes (must be a subset of the current right attribute names) or NULL.

Details

`toTable(x)` turns [Bimap](#) object `x` into a data frame (see section "Flat representation of a bimap" in `?Bimap` for a short introduction to this concept). For simple maps (i.e. no tags and no right attributes), the resulting data frame has only 2 columns, one for the left keys and one for the right keys, and each row in the data frame represents a link (or edge) between a left and a right key. For maps with tagged links (i.e. a tag is associated to each link), `toTable(x)` has one additional colmun for the tags and there is still one row per link. For maps with right attributes (i.e. a set of attributes is associated to each right key), `toTable(x)` has one additional colmun per attribute. So for example if `x` has tagged links and 2 right attributes, `toTable(x)` will have 5 columns: one for the left keys, one for the right keys, one for the tags, and one for each right attribute (always the rightmost columns). Note that if at least one of the right attributes is multivalued then more than 1 row can be needed to represent the same link so the number of rows in `toTable(x)` can be strictly greater than the number of links in the map.

`nrow(x)` is equivalent to (but more efficient than) `nrow(toTable(x))`.

`ncol(x)` is equivalent to (but more efficient than) `ncol(toTable(x))`.

`colnames(x)` is equivalent to (but more efficient than) `colnames(toTable(x))`. Columns are named accordingly to the names of the SQL columns where the data are coming from. An important consequence of this that they are not necessarily unique.

`colmetanames(x)` returns the metanames for the column of `x` that are not right attributes. Valid column metanames are `"Lkeyname"`, `"Rkeyname"` and `"tagname"`.

`Lkeyname`, `Rkeyname`, `tagname` and `Rattribnames` return the name of the column (or columns) containing the left keys, the right keys, the tags and the right attributes, respectively.

Like `toTable(x)`, `links(x)` turns `x` into a data frame but the right attributes (if any) are dropped. Note that dropping the right attributes produces a data frame that has eventually less columns than `toTable(x)` and also eventually less rows because now exactly 1 row is needed to represent 1 link.

`count.links(x)` is equivalent to (but more efficient than) `nrow(links(x))`.

`nhit(x)` returns a named integer vector indicating the number of "hits" for each key in `x` i.e. the number of links that start from each key.

Value

A data frame for `toTable` and `links`.

A single integer for `nrow`, `ncol` and `count.links`.

A character vector for `colnames`, `colmetanames` and `Rattribnames`.

A character string for `Lkeyname`, `Rkeyname` and `tagname`.

A named integer vector for `nhit`.

Author(s)

H. Pages

See Also

Bimap, BimapFormatting, AnnDbBimap-envirAPI

Examples

```
library(GO.db)
x <- GOSYNONYM
x
toTable(x)[1:4, ]
toTable(x["GO:0007322"])
links(x)[1:4, ]
links(x["GO:0007322"])

nrow(x)
ncol(x)
dim(x)
colnames(x)
colmetanames(x)
Lkeyname(x)
Rkeyname(x)
tagname(x)
Rattribnames(x)

links(x)[1:4, ]
count.links(x)

y <- GOBPCHILDREN
nhy <- nhit(y) # 'nhy' is a named integer vector
identical(names(nhy), keys(y)) # TRUE
table(nhy)
sum(nhy == 0) # number of GO IDs with no children
names(nhy)[nhy == max(nhy)] # the GO ID(s) with the most direct children

## Some sanity check
sum(nhy) == count.links(y) # TRUE

## Changing the right attributes of the GOSYNONYM map (advanced
## users only)
class(x) # GOTermsAnnDbBimap
as.list(x)[1:3]
colnames(x)
colmetanames(x)
```

```
tagname(x) # untagged map
Rattribnames(x)
Rattribnames(x) <- Rattribnames(x)[3:1]
colnames(x)
class(x) # AnnDbBimap
as.list(x)[1:3]
```

Bimap *Bimap objects and the Bimap interface*

Description

What we usually call "annotation maps" are in fact Bimap objects. In the following sections we present the bimap concept and the Bimap interface as it is defined in AnnotationDbi.

Display methods

In the code snippets below, `x` is a Bimap object.

`show(x):` Display minimal information about Bimap object `x`.

`summary(x):` Display a little bit more information about Bimap object `x`.

The bimap concept

A bimap is made of:

```
- 2 sets of objects: the left objects and the right objects.
  All the objects have a name and this name is unique in
  each set (i.e. in the left set and in the right set).
  The names of the left (resp. right) objects are called the
  left (resp. right) keys or the Lkeys (resp. the Rkeys).

- Any number of links (edges) between the left and right
  objects. Note that the links can be tagged. In our model,
  for a given bimap, either none or all the links are tagged.
```

In other words, a bimap is a bipartite graph.

Here are some examples:

```
1. bimap B1:

   4 left objects (Lkeys): "a", "b", "c", "d"
   3 objects on the right (Rkeys): "A", "B", "C"

   Links (edges):
    "a" <--> "A"
    "a" <--> "B"
    "b" <--> "A"
    "d" <--> "C"

   Note that:
     - There can be any number of links starting from or ending
       at a given object.
     - The links in this example are untagged.
```

```
2. bimap B2:

   4 left objects (Lkeys): "a", "b", "c", "d"
   3 objects on the right (Rkeys): "A", "B", "C"

   Tagged links (edges):
     "a" <-"x"-> "A"
     "a" <-"y"-> "B"
     "b" <-"x"-> "A"
     "d" <-"x"-> "C"
     "d" <-"y"-> "C"

   Note that there are 2 links between objects "d" and "C":
   1 with tag "x" and 1 with tag "y".
```

Flat representation of a bimap

The flat representation of a bimap is a data frame. For example, for B1, it is:

```
left   right
   a       A
   a       B
   b       A
   d       C
```

If in addition the right objects have 1 multivalued attribute, for example, a numeric vector:

```
A <-- c(1.2, 0.9)
B <-- character(0)
C <-- -1:1
```

then the flat representation of B1 becomes:

```
left   right   Rattrib1
   a       A        1.2
   a       A        0.9
   a       B         NA
   b       A        1.2
   b       A        0.9
   d       C         -1
   d       C          0
   d       C          1
```

Note that now the number of rows is greater than the number of links!

AnnDbBimap and FlatBimap objects

An AnnDbBimap object is a bimap whose data are stored in a data base. A FlatBimap object is a bimap whose data (left keys, right keys and links) are stored in memory (in a data frame for the links). Conceptually, AnnDbBimap and FlatBimap objects are the same (only their internal representation differ) so it's natural to try to define a set of methods that make sense for both (so they can be manipulated in a similar way). This common interface is the Bimap interface.

Note that both AnnDbBimap and FlatBimap objects have a read-only semantic: the user can subset them but cannot change their data.

The "flatten" generic

```
flatten(x) converts AnnDbBimap object x into FlatBimap
object y with no loss of information
```

Note that a FlatBimap object can't be converted into an AnnDbBimap object (well, in theory maybe it could be, but for now the data bases we use to store the data of the AnnDbBimap objects are treated as read-only). This conversion from AnnDbBimap to FlatBimap is performed by the "flatten" generic function (with methods for AnnDbBimap objects only).

Property0

The "flatten" generic plays a very useful role when we need to understand or explain exactly what a given Bimap method f will do when applied to an AnnDbBimap object. It's generally easier to explain what it does on a FlatBimap object and then to just say "and it does the same thing on an AnnDbBimap object". This is exactly what Property0 says:

```
for any AnnDbBimap object x, f(x) is expected to be
indentical to f(flatten(x))
```

Of course, this implies that the f method for AnnDbBimap objects return the same type of object than the f method for FlatBimap objects. In this sense, the "revmap" and "subset" Bimap methods are particular because they are expected to return an object of the same class as their argument x, so f(x) can't be identical to f(flatten(x)). For these methods, Property0 says:

```
for any AnnDbBimap object x, flatten(f(x)) is expected to
be identical to f(flatten(x))
```

Note to the AnnotationDbi maintainers/developpers: the `checkProperty0` function (AnnDbPkg-checker.R file) checks that Property0 is satisfied on all the AnnDbBimap objects defined in a given package (FIXME: checkProperty0 is currently broken).

The Bimap interface in AnnotationDbi

The full documentation for the methods of the Bimap interface is splitted into 4 man pages: Bimap-direction, Bimap-keys and Bimap-toTable.

See Also

Bimap-direction, Bimap-keys, Bimap-toTable, BimapFormatting, AnnDbBimap-envirAPI

Examples

```
library(hgu95av2.db)
ls(2)
hgu95av2GO # calls the "show" method
summary(hgu95av2GO)
hgu95av2GO2PROBE # calls the "show" method
summary(hgu95av2GO2PROBE)
```

BimapFormatting *Formatting a Bimap as a list or character vector*

Description

These functions format a Bimap as a list or character vector.

Usage

```
## Formatting as a list
as.list(x, ...)

## Formatting as a character vector
#as.character(x, ...)
```

Arguments

x   A [Bimap](#) object.

...   Further arguments are ignored.

Author(s)

H. Pages

See Also

[Bimap](#), [AnnDbBimap-envirAPI](#)

Examples

```
library(hgu95av2.db)
as.list(hgu95av2CHRLOC)[1:9]
as.list(hgu95av2ENTREZID)[1:9]
as.character(hgu95av2ENTREZID)[1:9]
```

GOTerms-class *Class "GOTerms"*

Description

A class to represent Gene Ontology nodes

Objects from the Class

Objects can be created by calls of the form `GOTerms(GOId, term, ontology, definition, synonym, secondary)`. GOId, term, and ontology are required.

Slots

GOID: Object of class `"character"` A character string for the GO id of a primary node.

Term: Object of class `"character"` A character string that defines the role of gene product corresponding to the primary GO id.

Ontology: Object of class `"character"` Gene Ontology category. Can be MF - molecular function, CC - cellular component, or BP - biological process.

Definition: Object of class `"character"` Further definition of the ontology of the primary GO id.

Synonym: Object of class `"character"` other ontology terms that are considered to be synonymous to the primary term attached to the GO id (e.g. "type I programmed cell death" is a synonym of "apoptosis"). Synonymous here can mean that the synonym is an exact synonym of the primary term, is related to the primary term, is broader than the primary term, is more precise than the primary term, or name is related to the term, but is not exact, broader or narrower.

Secondary: Object of class `"character"` GO ids that are secondary to the primary GO id as results of merging GO terms so that One GO id becomes the primary GO id and the rest become the secondary.

Methods

GOID `signature(object = "GOTerms")`: The get method for slot GOID.

Term `signature(object = "GOTerms")`: The get method for slot Term.

Ontology `signature(object = "GOTerms")`: The get method for slot Ontology.

Definition `signature(object = "GOTerms")`: The get method for slot Definition.

Synonym `signature(object = "GOTerms")`: The get method for slot Synonym.

Secondary `signature(object = "GOTerms")`: The get method for slot Secondary.

show `signature(x = "GOTerms")`: The method for pretty print.

Note

GOTerms objects are used to represent primary GO nodes in the SQLite-based annotation data package GO.db

References

http://www.geneontology.org/

Examples

```
gonode <- new("GOTerms", GOID="GO:1234567", Term="Test", Ontology="MF",
                        Definition="just for testing")
GOID(gonode)
Term(gonode)
Ontology(gonode)
```

---

make_eg_to_go_map   *Create GO to Entrez Gene maps for chip-based packages*

Description

Create a new map object mapping Entrez ID to GO (or vice versa) given a chip annotation data package.

This is a temporary solution until a more general pluggable map solution comes online.

Usage

```
make_eg_to_go_map(chip)
```

Arguments

chip   The name of the annotation data package.

Value

Either a `Go3AnnDbMap` or a `RevGo3AnnDbMap`.

Author(s)

Seth Falcon and Herve Pages

Examples

```
library("hgu95av2.db")

eg2go = make_eg_to_go_map("hgu95av2.db")
sample(eg2go, 2)

go2eg = make_go_to_eg_map("hgu95av2.db")
sample(go2eg, 2)
```

---

makeARABIDOPSISCHIP_DB

Creates a sqlite database, and then makes an annotation package with it

---

Description

This function 1st creates a SQLite file useful for making a SQLite based annotation package by using the correct popXXXCHIP_DB function. Next, this function produces an annotation package featuring the sqlite database produced.

Usage

```
makeARABIDOPSISCHIP_DB(affy, prefix, fileName, chipMapSrc, chipSrc,
outputDir, version, manufacturer, chipName, manufacturerUrl, author,
maintainer)
```

Arguments

`affy`	Boolean to indicate if this is starting from an affy csv file or not. If it is, then that will be parsed to make the sqlite file, if not, then you can feed a tab delimited file with IDs as was done before with AnnBuilder.
`prefix`	prefix is the first part of the eventual desired package name. (ie. "prefix.db")
`fileName`	The path and filename for the file to be parsed. This field can be ommitted if affy is set to TRUE. For all other arabidopsis chips, the IDs that match to the probes MUST be TAIR IDs.
`chipMapSrc`	The path and filename to the intermediate database containing the mapping data for allowed ID types and how these IDs relate to each other.
`chipSrc`	The path and filename to the intermediate database containing the annotation data for the sqlite to build.
`outputDir`	Where you would like the output files to be placed.
`version`	What is the version number for the desired package.
`manufacturer`	Who made the chip being described.
`chipName`	What is the name of the chip.
`manufacturerUrl`	
	URL for manufacturers website.
`author`	List of authors involved in making the package.
`maintainer`	List of package maintainers with email addresses for contact purposes.

Examples

```
## Not run:

## End(Not run)
```

`makeHUMANCHIP_DB`	*Creates a sqlite database, and then makes an annotation package with it*

Description

This function 1st creates a SQLite file useful for making a SQLite based annotation package by using the correct popXXXCHIP_DB function. Next, this function produces an annotation package featuring the sqlite database produced.

Usage

```
makeHUMANCHIP_DB(affy, prefix, fileName, otherSrc, chipMapSrc, chipSrc,
baseMapType, outputDir, version, manufacturer, chipName,
manufacturerUrl, author, maintainer)
```

Arguments

`affy`	Boolean to indicate if this is starting from an affy csv file or not. If it is, then that will be parsed to make the sqlite file, if not, then you can feed a tab delimited file with IDs as was done before with AnnBuilder.
`prefix`	prefix is the first part of the eventual desired package name. (ie. "prefix.db")
`fileName`	The path and filename for the file to be parsed. This can either be an affy csv file or it can be a more classic file type.
`otherSrc`	The path and filenames to any other lists of IDs which might add information about how a probe will map.
`chipMapSrc`	The path and filename to the intermediate database containing the mapping data for allowed ID types and how these IDs relate to each other.
`chipSrc`	The path and filename to the intermediate database containing the annotation data for the sqlite to build.
`baseMapType`	The type of ID that is used for the initial base mapping. If using a classic base mapping file, this should be the ID type present in the fileName. This can be any of the following values: "gb" = for genbank IDs "ug" = unigene IDs "eg" = Entrez Gene IDs "refseq" = refseq IDs "gbNRef" = mixture of genbank and refseq IDs
`outputDir`	Where you would like the output files to be placed.
`version`	What is the version number for the desired package.
`manufacturer`	Who made the chip being described.
`chipName`	What is the name of the chip.
`manufacturerUrl`	
	URL for manufacturers website.
`author`	List of authors involved in making the package.
`maintainer`	List of package maintainers with email addresses for contact purposes.

Examples

```
## Not run:
makeHUMANCHIP_DB(affy = TRUE,
                 prefix = "hgu95av2",
                 fileName = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/hgu95av2/HG_U
                 otherSrc = c(
                   EA="/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/hgu95av2/hgu95av2.E
                   UMICH="/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/hgu95av2/hgu95av
                 chipMapSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/chipm
                 chipSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/chipsrc_
                 baseMapType = "gbNRef",
                 version = "1.0.0",
                 manufacturer = "Affymetrix",
                 chipName = "hgu95av2",
                 manufacturerUrl = "http://www.affymetrix.com")
## End(Not run)
```

---

makeYEASTCHIP_DB *Creates a sqlite database, and then makes an annotation package with it*

Description

This function 1st creates a SQLite file useful for making a SQLite based annotation package by using the correct popXXXCHIP_DB function. Next, this function produces an annotation package featuring the sqlite database produced.

Usage

```
makeYEASTCHIP_DB(affy, prefix, fileName, chipSrc, outputDir, version,
manufacturer, chipName, manufacturerUrl, author, maintainer)
```

Arguments

affy	Boolean to indicate if this is starting from an affy csv file or not. If it is, then that will be parsed to make the sqlite file, if not, then you can feed a tab delimited file with IDs as was done before with AnnBuilder.
prefix	prefix is the first part of the eventual desired package name. (ie. "prefix.db")
fileName	The path and filename for the file to be parsed. This can either be an affy csv file or it can be a more classic file type.
chipSrc	The path and filename to the intermediate database containing the annotation data for the sqlite to build.
outputDir	Where you would like the output files to be placed.
version	What is the version number for the desired package.
manufacturer	Who made the chip being described.
chipName	What is the name of the chip.
manufacturerUrl	
	URL for manufacturers website.
author	List of authors involved in making the package.
maintainer	List of package maintainers with email addresses for contact purposes.

Examples

```
## Not run:

## End(Not run)
```

popARABIDOPSISCHIPDB

Populates an SQLite DB with and produces a schema definition

Description

Creates SQLite file useful for making a SQLite based annotation package. Also produces the
schema file which details the schema for the database produced.

Usage

```
popARABIDOPSISCHIPDB(affy, prefix, fileName, chipMapSrc, chipSrc, metaDataSrc,
outputDir, printSchema)
```

Arguments

affy	Boolean to indicate if this is starting from an affy csv file or not. If it is, then that will be parsed to make the sqlite file, if not, then you can feed a tab delimited file with IDs as was done before with AnnBuilder.
prefix	prefix is the first part of the eventual desired package name. (ie. "prefix.sqlite")
fileName	The path and filename for the file to be parsed. This field can be ommitted if affy is set to TRUE. For all other arabidopsis chips, the IDs that match to the probes MUST be TAIR IDs.
chipMapSrc	The path and filename to the intermediate database containing the mapping data for allowed ID types and how these IDs relate to each other.
chipSrc	The path and filename to the intermediate database containing the annotation data for the sqlite to build.
metaDataSrc	Either a named character vector containing pertinent information about the metadata OR the path and filename to the intermediate database containing the metadata information for the package.
	If this is a custom package, then it must be a named vector with the following fields:
	metaDataSrc <- c(DBSCHEMA="the DB schema", ORGANISM="the organism", SPECIES="the species", MANUFACTURER="the manufacturer", CHIPNAME="the chipName", MANUFACTURERURL="the manufacturerUrl")
outputDir	Where you would like the output files to be placed.
printSchema	Boolean to indicate whether or not to produce an output of the schema (default is FALSE).

Examples

```
## Not run:
  ##Set up the metadata
  my_metaDataSrc <- c( DBSCHEMA="the DB schema",
                    ORGANISM="the organism",
                    SPECIES="the species",
                    MANUFACTURER="the manufacturer",
                    CHIPNAME="the chipName",
                    MANUFACTURERURL="the manufacturerUrl")
```

```
  ##Builds the ag sqlite:
  popARABIDOPSISCHIPDB(affy = TRUE,
                       prefix = "ag",
                       fileName = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/ag/AG_a
                       chipMapSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db
                       chipSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/ch
                       metaDataSrc = my_metaDataSrc,
                       printSchema=TRUE)

  ##Or if the package is a standard package (it probably isn't):
  popARABIDOPSISCHIPDB(affy = TRUE,
                       prefix = "ag",
                       fileName = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/ag/AG_a
                       chipMapSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db
                       chipSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/ch
                       metaDataSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/c
                       printSchema = TRUE)
## End(Not run)
```

---

popHUMANCHIPDB	*Populates an SQLite DB with and produces a schema definition*

Description

Creates SQLite file useful for making a SQLite based annotation package. Also produces the schema file which details the schema for the database produced.

Usage

```
    popHUMANCHIPDB(affy, prefix, fileName, chipMapSrc, chipSrc, metaDataSrc,
    otherSrc, baseMapType, outputDir, printSchema)
```

Arguments

affy	Boolean to indicate if this is starting from an affy csv file or not. If it is, then that will be parsed to make the sqlite file, if not, then you can feed a tab delimited file with IDs as was done before with AnnBuilder.
prefix	prefix is the first part of the eventual desired package name. (ie. "prefix.sqlite")
fileName	The path and filename for the file to be parsed. This can either be an affy csv file or it can be a more classic file type.
chipMapSrc	The path and filename to the intermediate database containing the mapping data for allowed ID types and how these IDs relate to each other.
chipSrc	The path and filename to the intermediate database containing the annotation data for the sqlite to build.
metaDataSrc	Either a named character vector containing pertinent information about the metadata OR the path and filename to the intermediate database containing the metadata information for the package.
	If this is a custom package, then it must be a named vector with the following fields:

metaDataSrc <- c(DBSCHEMA="the DB schema", ORGANISM="the organism", SPECIES="the species", MANUFACTURER="the manufacturer", CHIP-
NAME="the chipName", MANUFACTURERURL="the manufacturerUrl")

otherSrc   The path and filenames to any other lists of IDs which might add information
                about how a probe will map.

baseMapType   The type of ID that is used for the initial base mapping. If using a classic base
                mapping file, this should be the ID type present in the fileName. This can be
                any of the following values: "gb" = for genbank IDs "ug" = unigene IDs "eg"
                = Entrez Gene IDs "refseq" = refseq IDs "gbNRef" = mixture of genbank and
                refseq IDs

outputDir   Where you would like the output files to be placed.

printSchema   Boolean to indicate whether or not to produce an output of the schema (default
                is FALSE).

Examples

```
## Not run:
  ##Set up the metadata
  my_metaDataSrc <- c( DBSCHEMA="the DB schema",
                       ORGANISM="the organism",
                       SPECIES="the species",
                       MANUFACTURER="the manufacturer",
                       CHIPNAME="the chipName",
                       MANUFACTURERURL="the manufacturerUrl")

  ##Builds the org.Hs.eg sqlite:
  popHUMANCHIPDB(affy=TRUE,
                 prefix="hgu95av2",
                 fileName="/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/hgu95av2/HG_U95
                 chipMapSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/chipm
                 chipSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/chipsrc_
                 metaDataSrc = my_metaDataSrc,
                 otherSrc=c(
                   EA="/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/hgu95av2/hgu95av2.E
                   UMICH="/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/hgu95av2/hgu95av
                 printSchema=TRUE)

  ##Or if the package is a standard package (it probably isn't):
  popHUMANCHIPDB(affy=TRUE,
                 prefix="hgu95av2",
                 fileName="/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/hgu95av2/HG_U95
                 chipMapSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/chipm
                 chipSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/chipsrc_
                 metaDataSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/meta
                 otherSrc=c(
                   EA="/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/hgu95av2/hgu95av2.E
                   UMICH="/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/hgu95av2/hgu95av
                 printSchema=TRUE)
## End(Not run)
```

---

popHUMANDB	*Populates an SQLite DB with and produces a schema definition*

Description

Creates SQLite file useful for making a SQLite based annotation package. Also produces the schema file which details the schema for the database produced.

Usage

```
popHUMANDB(prefix, chipSrc, metaDataSrc, outputDir, printSchema)
```

Arguments

`prefix`	prefix is the first part of the eventual desired package name. (ie. "prefix.sqlite")
`chipSrc`	The path and filename to the intermediate database containing the annotation data for the sqlite to build.
`metaDataSrc`	Either a named character vector containing pertinent information about the metadata OR the path and filename to the intermediate database containing the metadata information for the package.
	If this is a custom package, then it must be a named vector with the following fields:
	metaDataSrc <- c(DBSCHEMA="the DB schema", ORGANISM="the organism", SPECIES="the species", MANUFACTURER="the manufacturer", CHIPNAME="the chipName", MANUFACTURERURL="the manufacturerUrl")
`outputDir`	Where you would like the output files to be placed.
`printSchema`	Boolean to indicate whether or not to produce an output of the schema (default is FALSE).

Examples

```
## Not run:
  ##Set up the metadata
  my_metaDataSrc <- c( DBSCHEMA="the DB schema",
                   ORGANISM="the organism",
                   SPECIES="the species",
                   MANUFACTURER="the manufacturer",
                   CHIPNAME="the chipName",
                   MANUFACTURERURL="the manufacturerUrl")

  ##Builds the org.Hs.eg sqlite:
  popHUMANDB(prefix="org.Hs.eg",
            chipSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/chipsrc_huma
            metaDataSrc = my_metaDataSrc,
            printSchema=TRUE)

  ##Or if the package is a standard package (it probably isn't):
  ##Builds the org.Hs.eg sqlite:
  popHUMANDB(prefix="org.Hs.eg",
            chipSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/chipsrc_huma
            metaDataSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/metadata
```

```
                  printSchema=TRUE)
  ## End(Not run)
```

---

popYEASTCHIPDB   *Populates an SQLite DB with and produces a schema definition*

Description

Creates SQLite file useful for making a SQLite based annotation package. Also produces the schema file which details the schema for the database produced.

Usage

```
   popYEASTCHIPDB(affy, prefix, fileName, chipSrc, metaDataSrc,
   outputDir, printSchema)
```

Arguments

affy	Boolean to indicate if this is starting from an affy csv file or not. If it is, then that will be parsed to make the sqlite file, if not, then you can feed a tab delimited file with IDs as was done before with AnnBuilder.
prefix	prefix is the first part of the eventual desired package name. (ie. "prefix.sqlite")
fileName	The path and filename for the file to be parsed. This can either be an affy csv file or it can be a more classic file type.
chipSrc	The path and filename to the intermediate database containing the annotation data for the sqlite to build.
metaDataSrc	Either a named character vector containing pertinent information about the meta-data OR the path and filename to the intermediate database containing the meta-data information for the package.
	If this is a custom package, then it must be a named vector with the following fields:
	metaDataSrc <- c(DBSCHEMA="the DB schema", ORGANISM="the organism", SPECIES="the species", MANUFACTURER="the manufacturer", CHIP-NAME="the chipName", MANUFACTURERURL="the manufacturerUrl")
outputDir	Where you would like the output files to be placed.
printSchema	Boolean to indicate whether or not to produce an output of the schema (default is FALSE).

Examples

```
## Not run:
  ##Set up the metadata
  my_metaDataSrc <- c( DBSCHEMA="the DB schema",
                  ORGANISM="the organism",
                  SPECIES="the species",
                  MANUFACTURER="the manufacturer",
                  CHIPNAME="the chipName",
                  MANUFACTURERURL="the manufacturerUrl")

  ##Builds the ag sqlite:
```

```
popYEASTCHIPDB(affy=TRUE,
               fileName="/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/yeast2/Yeast_2_
               prefix="yeast2",
               chipSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/chipsrc_
               metaDataSrc = my_metaDataSrc,
               printSchema=TRUE)

    ##Or if the package is a standard package (it probably isn't):
    popYEASTCHIPDB(affy=TRUE,
               fileName="/mnt/cpb_anno/mcarlson/proj/sqliteGen/srcFiles/yeast2/Yeast_2_
               prefix="yeast2",
               chipSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/chipsrc_
               metaDataSrc = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/meta
               printSchema=TRUE)
## End(Not run)
```

---

wrapBaseDBPackages *Wrap up all the Base Databases into Packages for distribution*

Description

Creates extremely simple packages from the base database files for distribution. This is a convenience function for wrapping up these packages in a consistent way each time.

Usage

```
wrapBaseDBPackages(dbPath, destDir, version)
```

Arguments

dbPath   dbPath is just the path to the location of the latest intermediate sqlite source files.
            These files are then used to make base DB packages.

destDir   destination path for the newly minted packages.

version   version number to stamp onto these newly minted packages.

Examples

```
## Not run:
  ##Make all of the intermediate DBs and place the new packages right here.
  wrapBaseDBPackages(dbPath = "/mnt/cpb_anno/mcarlson/proj/sqliteGen/nli/annosrc/db/",
                     destDir = ".")
## End(Not run)
```

`toSQLStringSet`	*Convert a vector to a quoted string for use as a SQL value list*

Description

Given a vector, this function returns a string with each element of the input coerced to character, quoted, and separated by ",".

Usage

```
toSQLStringSet(names)
```

Arguments

`names`   A vector of values to quote

Details

If `names` is a character vector with elements containing single quotes, these quotes will be doubled so as to escape the quote in SQL.

Value

A character vector of length one that represents the input vector as a SQL value list. Each element is single quoted and elements are comma separated.

Note

Do not use `sQuote` for generating SQL as that function is intended for display purposes only. In some locales, `sQuote` will generate fancy quotes which will break your SQL.

Author(s)

Herve Pages

Examples

```
toSQLStringSet(letters[1:4])
toSQLStringSet(c("'foo'", "ab'cd", "bar"))
```

---

unlist2   *A replacement for unlist() that does not mangle the names*

Description

`unlist2` is a replacement for `base::unlist()` that does not mangle the names.

Usage

```
unlist2(x, recursive=TRUE, use.names=TRUE, what.names="inherited")
```

Arguments

x, recursive, use.names
         See `?unlist`.

what.names   `"inherited"` or `"full"`.

Details

Use this function if you don't like the mangled names returned by the standard `unlist` function from the base package. Using `unlist` with annotation data is dangerous and it is highly recommended to use `unlist2` instead.

Author(s)

Herve Pages

See Also

[unlist](#)

Examples

```
x <- list(A=c(b=-4, 2, b=7), B=3:-1, c(a=1, a=-2), C=list(c(2:-1, d=55), e=99))
unlist(x)
unlist2(x)

library(hgu95av2.db)
egids <- c("10", "100", "1000")
egids2pbids <- mget(egids, revmap(hgu95av2ENTREZID))
egids2pbids

unlist(egids2pbids)    # 1001, 1002, 10001 and 10002 are not real
                       # Entrez ids but are the result of unlist()
                       # mangling the names!

unlist2(egids2pbids)   # much cleaner! yes the names are not unique
                       # but at least they are correct...
```

Index