Package 'xegaDerivationTrees'

February 13, 2024

Title Generating and Manipulating Derivation Trees

Version 1.0.0.0

Description Derivation tree operations are needed for implementing
grammar-based genetic programming and grammatical evolution:
Generating of a random derivation trees of a context-free grammar
of bounded depth, decoding a derivation tree,
choosing a random node in a derivation tree,
extracting a tree whose root is a specified node, and
inserting a subtree into a derivation tree at a specified node.
These operations are necessary for the initializiation and
for decoders of a random population of programs,
as well as for implementing crossover and mutation operators.
Depth-bounds are guaranteed by switching to a grammar
without recursive production rules.
For executing the examples, the package 'BNF' is needed.
The basic tree operations of generating, extracting, and
inserting of derivation trees as well as the conditions
for guaranteeing complete derivation trees have been
presented in Geyer-Schulz (1997, ISBN:978-3-7908-0830-X).
The use of random integer vectors for the generation
of derivation trees has been introduced in
Ryan, C., Collins, J. J., and O'Neill, M. (1998)
<doi:10.1007/BFb0055930>.

License MIT + file LICENSE

URL <https://github.com/ageyerschulz/xegaDerivationTrees>

Encoding UTF-8

RoxygenNote 7.2.3

Suggests testthat (>= 3.0.0)

Imports xegaBNF

NeedsCompilation no

Author Andreas Geyer-Schulz [aut, cre]
(<https://orcid.org/0009-0000-5237-3579>)

Maintainer Andreas Geyer-Schulz <Andreas.Geyer-Schulz@kit.edu>

Repository CRAN

Date/Publication 2024-02-13 16:50:02 UTC

R topics documented:

`booleanGrammar` *A constant function which returns the BNF (Backus-Naur Form) of a context-free grammar for the XOR problem.*

Description

A constant function which returns the BNF (Backus-Naur Form) of a context-free grammar for the XOR problem.

Usage

```
booleanGrammar()
```

Details

Imported from package xegaBNF for use in examples.

Value

A named list with elements `$filename` and `$BNF` representing the grammar of a boolean grammar with two variables and the boolean functions `AND`, `OR`, and `NOT`.

See Also

Other Grammar: `compileBNF()`

Examples

```
booleanGrammar()
```

---

chooseNode *Selects an attributed node in an attributed node list randomly.*

Description

`chooseNode()` returns a random attributed node from an attributed node list

Usage

```
chooseNode(ANL)
```

Arguments

`ANL`	Attributed node list.

Details

An attributed node has the following elements:

- `ID`
- `NonTerminal`
- `Pos`
- `Depth`
- `Rdepth`
- `subtreedepth`
- `node$Index`

These elements can be used e.g.

- for inserting and extracting subtrees (`Pos` or `node$Index`),
- for checking the feasibility of subtree substitution (`ID`),
- for checking depth bounds (`Depth`, `RDepth`, and `subtreedepth`), ...

Value

Attributed node.

See Also

Other Random Choice: `chooseRule()`

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
b<-treeANL(a, g$ST)
c<-chooseNode(b$ANL)
```

---

chooseRule *Selects a production rule index at random from a vector of production rules.*

Description

`chooseRule()` selects a production rule index from the vector of production rule indices in the `g$PT$LHS$` for a non-terminal symbol.

Usage

```
chooseRule(riv)
```

Arguments

`riv`	Vector of production rules indices for a non-terminal symbol.

Value

Integer. Index of the production rule.

See Also

Other Random Choice: `chooseNode()`

Examples

```
chooseRule(c(7, 8, 9))
chooseRule(as.vector(1))
```

chooseRulek *Selects k-th production rule index from a vector of production rules.*

Description

`chooseRulek()` selects the k-th production rule index from the vector of production rule indices in the `g$PT$LHS$` for a non-terminal symbol.

Usage

```
chooseRulek(riv, k)
```

Arguments

`riv`	Vector of production rules indices for a non-terminal symbol.
`k`	Integer.

Value

The index of the production rule.

Examples

```
chooseRulek(c(7, 8, 9), 9)
chooseRulek(as.vector(1), 9)
```

compatibleSubtrees *Test the compatibility of subtrees.*

Description

`compatibleSubtrees()` tests the compatibility of two subtrees.

Usage

```
compatibleSubtrees(n1, n2, maxdepth = 5, DepthBounded = TRUE)
```

Arguments

`n1`	Attributed node of the root of subtree 1.
`n2`	Attributed node of the root of subtree 2.
`maxdepth`	Integer. Maximal derivation depth.
`DepthBounded`	• `TRUE`: Only subtrees with the same root symbol and which respect the depth restrictions are compatible. • `FALSE`: The depth restrictions are not checked.

Details

`compatibleSubtrees()` tests the compatibility of two subtrees:

1. The root symbol of the two subtrees must be identical: `(n1$ID==n2$ID)`.
2. The depth restrictions must hold:
   (a) `depth(n1) + depth(subtree2) <= maxdepth+maxSPT`
   (b) `depth(n2) + depth(subtree1) <= maxdepth+maxSPT`

   maxSPT is the maximal number of derivations needed to generate a complete derivation tree.

Value

`TRUE` or `FALSE`

See Also

Other Tree Operations: `treeExtract()`, `treeInsert()`

Examples

```
g<-compileBNF(booleanGrammar())
t1<-randomDerivationTree(g$Start, g)
t1anl<-treeANL(t1, g$ST)
t2<-randomDerivationTree(g$Start, g)
t2anl<-treeANL(t2, g$ST)
n1<-chooseNode(t1anl$ANL)
n2<-chooseNode(t2anl$ANL)
compatibleSubtrees(n1, n2)
compatibleSubtrees(n1, n2, maxdepth=1)
compatibleSubtrees(n1, n2, DepthBounded=FALSE)
```

---

compileBNF *Compile a BNF (Backus-Naur Form) of a context-free grammar.*

Description

`compileBNF()` produces a context-free grammar from its specification in Backus-Naur form (BNF). Warning: No error checking is implemented.

Usage

```
compileBNF(g, verbose = FALSE)
```

Arguments

`g`	A character string with a BNF.
`verbose`	Boolean. TRUE: Show progress. Default: FALSE.

Details

A grammar consists of the symbol table `ST`, the production table `PT`, the start symbol `Start`, and the short production table `SPT`.

The function performs the following steps:

1. Make the symbol table.
2. Make the production table.
3. Extract the start symbol.
4. Compile a short production table.
5. Return the grammar.

Value

A grammar object (list) with the attributes

- `name`: Filename of the grammar.
- `ST`: Symbol table.
- `PT`: Production table.
- `Start`: Start symbol of the grammar.
- `SPT`: Short production table.

See Also

Other Grammar: `booleanGrammar()`

Examples

```
g<-compileBNF(booleanGrammar())
g$ST
g$PT
g$Start
g$SPT
```

---

decodeCDT *Converts a complete derivation tree into a program.*

Description

`decodeCDT()` returns a program (a text string with the terminal symbol string). If the derivation tree still has non-terminal leaves, the non-terminal leaves are omitted. The program produces a syntax error. The program can not be repaired.

Usage

```
decodeCDT(tree, ST)
```

Arguments

tree	Derivation tree.
ST	Symbol table.

Value

Program.

See Also

Other Decoder: decodeDTsym(), decodeDT(), decodeTree(), leavesIncompleteDT()

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
decodeCDT(a, g$ST)
```

---

decodeDT *Decodes a derivation tree into a program.*

Description

The program may contain non-terminal symbols and its evaluation may fail.

Usage

```
decodeDT(tree, ST)
```

Arguments

tree	Derivation tree.
ST	Symbol table.

Value

Program

See Also

Other Decoder: decodeCDT(), decodeDTsym(), decodeTree(), leavesIncompleteDT()

Examples

```
g<-compileBNF(booleanGrammar())
t1<-generateDerivationTree(sym=g$Start,sample(100, 10, replace=TRUE), G=g)
decodeDT(t1$tree, g$ST)
```

`decodeDTsym` *Decodes a derivation tree into a list of the leaf symbols of the derivation tree.*

Description

Decodes a derivation tree into a list of the leaf symbols of the derivation tree.

Usage

```
decodeDTsym(tree, ST)
```

Arguments

`tree`	Derivation tree.
`ST`	Symbol table.

Value

List of the leaf symbols of the derivation tree.

See Also

Other Decoder: `decodeCDT()`, `decodeDT()`, `decodeTree()`, `leavesIncompleteDT()`

Examples

```
g<-compileBNF(booleanGrammar())
t1<-generateDerivationTree(sym=g$Start,sample(100, 10, replace=TRUE), G=g)
decodeDTsym(t1$tree, g$ST)
```

`decodeTree` *Returns a list of all symbols of a derivation tree in depth-first left-to-right order.*

Description

`decodeTree()` returns a list of all symbols of a derivation tree in depth-first left-to-right order (coded as R Factor with the symbol identifiers as levels).

Usage

```
decodeTree(tree, ST)
```

Arguments

tree	Derivation tree.
ST	Symbol table.

Value

List of all symbols in depth-first left-to-right order.

See Also

Other Decoder: decodeCDT(), decodeDTsym(), decodeDT(), leavesIncompleteDT()

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
decodeTree(a, g$ST)
```

---

filterANL *Filter an Attributed Node List (ANL) of a derivation tree by depth.*

Description

filterANL() deletes all nodes whose depth node$Depth is less than minb and larger than maxb from the ANL. However, if the resulting list is empty, the original ANL is returned.

Usage

```
filterANL(ANL, minb = 1, maxb = 3)
```

Arguments

ANL	Attributed node list.
minb	Integer.
maxb	Integer.

Details

An attributed node has the following elements:

- $ID: Id in the symbol table ST.
- $NT: Is the symbol a non-terminal?
- $Pos: Position in the trail.
- $Depth: Depth of node.
- $RDepth: Residual depth for expansion.
- $subtreedepth: Depth of subtree starting here.
- $Index: R index of the node in the derivation tree. Allows fast tree extraction and insertion.

Value

An attributed node list with nodes whose depths are in `minb:maxb`. Each node is represented as a list of the following attributes:

- `Node$ID`: Id in the symbol table ST.
- `Node$NT`: Is the symbol a non-terminal?
- `Node$Pos`: Position in the trail.
- `Node$Depth`: Depth of node.
- `Node$RDepth`: Residual depth for expansion.
- `Node$subtreedepth`: Depth of subtree starting here.
- `Node$Index`: R index of the node in the derivation tree. Allows fast tree extraction and insertion.

See Also

Other Access Tree Parts: `filterANLid()`, `treeANL()`, `treeChildren()`, `treeRoot()`

Examples

```
g<-compileBNF(booleanGrammar())
set.seed(111)
a<-randomDerivationTree(g$Start, g, maxdepth=10)
b<-treeANL(a, g$ST)
c<-filterANL(b, minb=1, maxb=3)
d<-filterANL(b, minb=3, maxb=5)
e<-filterANL(b, minb=14, maxb=15)
f<-filterANL(b, minb=13, maxb=15)
```

---

filterANLid *Filter an Attributed Node List (ANL) of a derivation tree by a symbol identifier.*

Description

`filterANLid()` deletes all nodes whose `node$ID` does not match `node$ID`. If the resulting list is empty, a list of length 0 is returned.

Usage

```
filterANLid(ANL, nodeID = 1)
```

Arguments

`ANL`	Attributed node list.
`nodeID`	Integer. The identifier of a symbol.

Details

An attributed node has the following elements:

- `$ID`: Id in the symbol table `ST`.
- `$NT`: Is the symbol a non-terminal?
- `$Pos`: Position in the trail.
- `$Depth`: Depth of node.
- `$RDepth`: Residual depth for expansion.
- `$subtreedepth`: Depth of subtree starting here.
- `$Index`: R index of the node in the derivation tree. Allows fast tree extraction and insertion.

For the implementation of crossover and mutation, we expect a non-terminal symbol identifier.

Value

An attributed node list with nodes whose depths are in `minb:maxb`. Each node is represented as a list of the following attributes:

- `Node$ID`: Id in the symbol table ST.
- `Node$NT`: Is the symbol a non-terminal?
- `Node$Pos`: Position in the trail.
- `Node$Depth`: Depth of node.
- `Node$RDepth`: Residual depth for expansion.
- `Node$subtreedepth`: Depth of subtree starting here.
- `Node$Index`: R index of the node in the derivation tree. Allows fast tree extraction and insertion.

See Also

Other Access Tree Parts: `filterANL()`, `treeANL()`, `treeChildren()`, `treeRoot()`

Examples

```
g<-compileBNF(booleanGrammar())
set.seed(111)
a<-randomDerivationTree(g$Start, g, maxdepth=10)
b<-treeANL(a, g$ST)
c<-filterANLid(b, nodeID=5)
d<-filterANLid(b, nodeID=6)
e<-filterANLid(b, nodeID=7)
f<-filterANLid(b, nodeID=8)
```

`generateDerivationTree` *Generates a derivation tree from an integer vector.*

Description

`generateDerivationTree()` generates a derivation tree from an integer vector. The derivation tree may be incomplete.

Usage

```
generateDerivationTree(sym, kvec, complete = TRUE, G, maxdepth = 5)
```

Arguments

`sym`	Non-terminal symbol.
`kvec`	Integer vector.
`complete`	Boolean. FALSE for incomplete derivation trees.
`G`	Grammar.
`maxdepth`	Integer. Maximal depth of the derivation tree.

Details

`generateDerivationTree()` recursively expands non-terminals and builds a derivation tree.

Value

A named list l$tree, l$kvec, l$complete.

See Also

Other Generate Derivation Tree: `randomDerivationTree()`, `rndsubk()`, `rndsub()`, `substituteSymbol()`

Examples

```
g<-compileBNF(booleanGrammar())
a<-sample(100, 100, replace=TRUE)
b<-generateDerivationTree(sym=g$Start, kvec=a, G=g, maxdepth=10)
decodeDT(b$tree, g$ST)
```

`leavesIncompleteDT` *Returns the list of symbol identifiers of the leaves of a derivation tree.*

Description

For incomplete derivation trees, non-terminal symbols are leaves.

Usage

```
leavesIncompleteDT(tree, ST, leavesList = list())
```

Arguments

`tree`	Derivation tree.
`ST`	Symbol table.
`leavesList`	List of symbol identifiers.

Details

Must perform a depth-first left-to-right tree traversal to collect all leave symbols (terminal and non-terminal symbols).

Value

List of symbol identifiers.

See Also

Other Decoder: `decodeCDT()`, `decodeDTsym()`, `decodeDT()`, `decodeTree()`

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
leavesIncompleteDT(a, g$ST)
```

randomDerivationTree *Generates a random derivation tree.*

Description

randomDerivationTree() generates a random derivation tree.

Usage

```
randomDerivationTree(sym, G, maxdepth = 5, CompleteDT = TRUE)
```

Arguments

sym	Non-terminal symbol.
G	Grammar.
maxdepth	Integer. Maximal depth of the derivation tree.
CompleteDT	Boolean. Generate a complete derivation tree? Default: TRUE.

Details

RandomDerivationTree() recursively expands non-terminals and builds a depth-bounded derivation tree.

Value

Derivation tree (a nested list).

See Also

Other Generate Derivation Tree: generateDerivationTree(), rndsubk(), rndsub(), substituteSymbol()

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
b<-randomDerivationTree(g$Start, g, maxdepth=10)
c<-randomDerivationTree(g$Start, g, 2, FALSE)
```

rndPartition *Randomly partitions n in k parts.*

Description

Sampling a partition is a two-step process:

1. The k parts of the partion are sampled in the loop. This implies that the first partition p is a random number between 1 and 1+n-k. The next partition is sampled from 1 to 1+n-k-p.
2. We permute the partitions.

Usage

```
rndPartition(n, k)
```

Arguments

n	The integer to divide.
k	Number of parts.

Value

The integer partition of n in k parts.

Examples

```
rndPartition(10, 4)
```

rndsub *Transforms a non-terminal symbol into a random 1-level derivation tree.*

Description

`rndsub()` expands a non-terminal by a random derivation and returns a 1-level derivation tree.

Usage

```
rndsub(sym, PT)
```

Arguments

sym	Non-terminal symbol.
PT	Production table.

Value

Derivation tree with 1-level.

See Also

Other Generate Derivation Tree: `generateDerivationTree()`, `randomDerivationTree()`, `rndsubk()`, `substituteSymbol()`

Examples

```
g<-compileBNF(booleanGrammar())
rndsub(g$Start, g$PT)
```

`rndsubk` *Transforms a non-terminal symbol into a 1-level derivation tree for a given k.*

Description

`rndsubk()` expands a non-terminal by a derivation specified by k and returns a 1-level derivation tree.

Usage

```
rndsubk(sym, k, PT)
```

Arguments

`sym`	Non-terminal symbol.
`k`	Codon (An integer).
`PT`	Production table.

Value

1-level derivation tree.

See Also

Other Generate Derivation Tree: `generateDerivationTree()`, `randomDerivationTree()`, `rndsub()`, `substituteSymbol()`

Examples

```
g<-compileBNF(booleanGrammar())
rndsubk(g$Start, 207, g$PT)
```

`substituteSymbol` *Codes the substitution of a non-terminal symbol by the symbols derived by a production rule as a nested list.*

Description

`substituteSymbol()` generates a nested list with the non-terminal symbol as the root (first list element) and the derived symbols as the second list element.

Usage

```
substituteSymbol(rindex, PT)
```

Arguments

`rindex`	Rule index.
`PT`	Production table.

Value

2-element list.

See Also

Other Generate Derivation Tree: `generateDerivationTree()`, `randomDerivationTree()`, `rndsubk()`, `rndsub()`

Examples

```
g<-compileBNF(booleanGrammar())
substituteSymbol(3, g$PT)
```

`testGenerateDerivationTree` *Generate, decode, and show* `times` *derivation trees from random integer vectors for grammar BNF on the console.*

Description

Generate, decode, and show `times` derivation trees from random integer vectors for grammar BNF on the console.

Usage

```
testGenerateDerivationTree(times, BNF, verbose = TRUE)
```

Arguments

times	Number of derivation trees which should be generated.
BNF	BNF.
verbose	Boolean. If TRUE (default) , print decoded derivation tree on console.

Value

Number of complete derivation trees generated.

Examples

```
testGenerateDerivationTree(5, BNF=booleanGrammar())
```

---

treeANL *Builds an Attributed Node List (ANL) of a derivation tree.*

Description

`treeANL()` recursively traverses a derivation tree and collects information about the derivation tree in an attributed node list (ANL).

Usage

```
treeANL(
  tree,
  ST,
  maxdepth = 5,
  ANL = list(),
  IL = list(),
  count = 1,
  depth = 1
)
```

Arguments

tree	A derivation tree.
ST	A symbol table.
maxdepth	Limit on the depth of a derivation tree.
ANL	Attributed node list (empty on invocation).
IL	Index function list (empty on invocation).
count	Trail count (1 on invocation).
depth	Derivation tree depth (1 on invocation).

Details

An attributed node has the following elements:

- $ID: Id in the symbol table ST.
- $NT: Is the symbol a non-terminal?
- $Pos: Position in the trail.
- $Depth: Depth of node.
- $RDepth: Residual depth for expansion.
- $subtreedepth: Depth of subtree starting here.
- $Index: R index of the node in the derivation tree. Allows fast tree extraction and insertion.

These elements can be used e.g.

- for inserting and extracting subtrees (Pos or node$Index),
- for checking the feasibility of subtree substitution (ID),
- for checking depth bounds (Depth, RDepth, and subtreedepth), ...

Value

A list with three elements:

1. r$count: The trail length (not needed).
2. r$depth: The derivation tree depth (not needed).
3. r$ANL: The attributed node list is a list of nodes. Each node is represented as a list of the following attributes:
   - Node$ID: Id in the symbol table ST.
   - Node$NT: Is the symbol a non-terminal?
   - Node$Pos: Position in the trail.
   - Node$Depth: Depth of node.
   - Node$RDepth: Residual depth for expansion.
   - Node$subtreedepth: Depth of subtree starting here.
   - Node$Index: R index of the node in the derivation tree. Allows fast tree extraction and insertion.

See Also

Other Access Tree Parts: filterANLid(), filterANL(), treeChildren(), treeRoot()

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
b<-treeANL(a, g$ST)
c<-treeANL(a, g$ST, 10)
d<-treeANL(a, g$ST, maxdepth=10)
```

treeChildren *Returns the children of a derivation tree.*

Description

`treeChildren()` returns the children of a derivation tree represented as a list of derivation trees.

Usage

```
treeChildren(tree)
```

Arguments

`tree`	Derivation tree.

Value

The children of a derivation tree (a list of derivation trees).

See Also

Other Access Tree Parts: `filterANLid()`, `filterANL()`, `treeANL()`, `treeRoot()`

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
treeChildren(a)
```

treeExtract *Extracts the subtree at position* `pos` *in a derivation tree.*

Description

`treeExtract()` returns the subtree at position `pos` in a derivation tree.

Usage

```
treeExtract(tree, node)
```

Arguments

`tree`	Derivation tree.
`node`	Attributed node.

Details

An attributed node is a list whose element node$Index contains an access function to the node. The access function is represented as a string with an executable R index expression. All what remains to be done, is

- to complete the access statement and
- to return the result of parsing and evaluating the string.

Value

Derivation tree.

See Also

Other Tree Operations: compatibleSubtrees(), treeInsert()

Examples

```
g<-compileBNF(booleanGrammar())
t1<-randomDerivationTree(g$Start, g)
t1anl<-treeANL(t1, g$ST)
n1<-chooseNode(t1anl$ANL)
st1<-treeExtract(t1, n1)
decodeCDT(st1, g$ST)
st2<-treeExtract(t1, chooseNode(t1anl$ANLa))
decodeCDT(st2, g$ST)
```

---

treeInsert *Inserts a subtree into a derivation tree at a* node.

Description

treeInsert() inserts a subtree into a tree at a node.

Usage

```
treeInsert(tree, subtree, node)
```

Arguments

tree	Derivation tree.
subtree	Subtree.
node	Attributed node.

Details

An attributed node is a list whose element node$Index contains an access function to the node. The access function is represented as a string which contains an executable R index expression. All what remains to be done, is

- to complete the assignment statement and
- to parse and evaluate the string.

Value

A derivation tree.

See Also

Other Tree Operations: compatibleSubtrees(), treeExtract()

Examples

```
g<-compileBNF(booleanGrammar())
t1<-randomDerivationTree(g$Start, g)
t2<-randomDerivationTree(g$Start, g)
t1anl<-treeANL(t1, g$ST)
n1<-chooseNode(t1anl$ANL)
t2<-randomDerivationTree(n1$ID, g)
tI1<-treeInsert(t1, t2, n1)
decodeCDT(tI1, g$ST)
```

---

treeLeaves *Measures the number of leaves of a complete derivation tree.*

Description

treeLeaves() returns the number of terminal symbols in a complete derivation tree.

Usage

```
treeLeaves(tree, ST)
```

Arguments

tree	Derivation tree.
ST	Symbol table.

Value

Integer. Number of terminal symbols in a complete derivation tree.

See Also

Other Measures of Tree Attributes: `treeListDepth()`, `treeNodes()`, `treeSize()`

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
treeLeaves(a, g$ST)
((treeLeaves(a, g$ST)+treeNodes(a, g$ST)) == treeSize(a))
```

---

treeListDepth *Measures the depth of a (nested) list.*

Description

`treeListDepth()` returns the depth of a nested list. For a derivation tree, this is approximately twice the derivation depth.

Usage

```
treeListDepth(t, tDepth = 0)
```

Arguments

`t`	List.
`tDepth`	Integer. List depth. Default: 0.

Value

Depth of a nested list.

See Also

Other Measures of Tree Attributes: `treeLeaves()`, `treeNodes()`, `treeSize()`

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
treeListDepth(a)
```

treeNodes *Measures the number of inner nodes in a derivation tree.*

Description

`treeNodes()` returns the number of non-terminal symbols in a derivation tree.

Usage

```
treeNodes(tree, ST)
```

Arguments

`tree`	Derivation tree.
`ST`	Symbol table.

Value

Integer. Number of non-terminal symbols in a derivation tree.

See Also

Other Measures of Tree Attributes: `treeLeaves()`, `treeListDepth()`, `treeSize()`

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
treeNodes(a, g$ST)
```

treeRoot *Returns the root of a derivation tree.*

Description

`treeRoot()` returns the root of a derivation tree.

Usage

```
treeRoot(tree)
```

Arguments

`tree`	Derivation tree.

Value

Root of a derivation tree.

See Also

Other Access Tree Parts: `filterANLid()`, `filterANL()`, `treeANL()`, `treeChildren()`

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
treeRoot(a)
```

---

treeSize *Measures the number of symbols in a derivation tree.*

Description

`treeSize()` returns the number of symbols in a derivation tree.

Usage

```
treeSize(tree)
```

Arguments

`tree`	Derivation tree.

Value

Integer. Number of symbols in a derivation tree.

See Also

Other Measures of Tree Attributes: `treeLeaves()`, `treeListDepth()`, `treeNodes()`

Examples

```
g<-compileBNF(booleanGrammar())
a<-randomDerivationTree(g$Start, g)
treeSize(a)
```

xegaDerivationTrees *Package xegaDerivationTrees*

Description

Derivation Trees

Details

The implementation of a data type for derivation trees.

The derivation tree operations for generating complete random subtrees and for for subtree extraction and insertion are formally introduced in Geyer-Schulz (1997) and used for implementing mutation and crossover operations.

Efficient selection of random subtrees is implemented by building a list of annotated tree nodes by a left-right depth-first tree traversal. For each node, the R-index to access the subtree is built and stored in the node. The R-index element of a node allows subtree extraction and insertion operations with the cost of the R-index operation. In addition, filtering operations the node list by different criteria (min depth, max depth, and non-terminal symbol type) allow the implementation of flexible and configurable crossover and mutation operations.

The Architecture of the xegaX-Packages

The xegaX-packages are a family of R-packages which implement eXtended Evolutionary and Genetic Algorithms (xega). The architecture has 3 layers, namely the user interface layer, the population layer, and the gene layer:

- The user interface layer (package `xega`) provides a function call interface and configuration support for several algorithms: genetic algorithms (sga), permutation-based genetic algorithms (sgPerm), derivation-free algorithms as e.g. differential evolution (sgde), grammar-based genetic programming (sgp) and grammatical evolution (sge).
- The population layer (package `xegaPopulation`) contains population-related functionality as well as support for adaptive mechanisms which depend on population statistics. In addition, support for parallel evaluation of genes is implemented here.
- The gene layer is split in a representation-independent and a representation-dependent part:
  1. The representation-indendent part (package `xegaSelectGene`) is responsible for variants of selection operators, evaluation strategies for genes, as well as profiling and timing capabilities.
  2. The representation-dependent part consists of the following packages:
     - `xegaGaGene` for binary-coded genetic algorithms.
     - `xegaPermGene` for permutation-based genetic algorithms.
     - `xegaDfGene` for derivation-free algorithms. For example, differential evolution.
     - `xegaGpGene` for grammar-based genetic algorithms.
     - `xegaGeGene` for grammatical evolution algorithms.

     The packages `xegaDerivationTrees` and `xegaBNF` support the last two packages: `xegaBNF` essentially provides a grammar compiler, and `xegaDerivationTrees` an abstract data type for derivation trees.

Copyright

License

MIT

URL

<https://github.com/ageyerschulz/xegaDerivationTrees>

Installation

From cran with `install.packages("xegaDerivationTrees")`

Author(s)

Andreas Geyer-Schulz

References

Geyer-Schulz, Andreas (1997): *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning*, Physica, Heidelberg. (ISBN:978-3-7908-0830-X)

Index