Package 'xegaBNF'

February 5, 2024

Title Compile a Backus-Naur Form Specification into an R Grammar Object

Version 1.0.0.0

Description Translates a BNF (Backus-Naur Form) specification of a context-free language into an R grammar object which consists of the start symbol, the symbol table, the production table, and a short production table. The short production table is non-recursive. The grammar object contains the file name from which it was generated (without a path). In addition, it provides functions to determine the type of a symbol (isTerminal() and isNonterminal()) and functions to access the production table (rules() and derives()). For the BNF specification, see Backus, John et al. (1962) ``Revised Report on the Algorithmic Language ALGOL 60''. (ALGOL60 standards page <http://www.algol60.org/2standards.htm>, html-edition <https://www.masswerk.at/algol60/report.htm>) The grammar compiler is based on the APL2 implementation in Geyer-Schulz, Andreas (1997, ISBN:978-3-7908-0830-X).

License MIT + file LICENSE

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

Encoding UTF-8

RoxygenNote 7.2.3

Suggests testthat (>= 3.0.0)

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 2024-02-05 20:50:09 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()
```

Value

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

Examples

```
booleanGrammar()
```

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. See makeSymbolTable.
2. Make the production table. See makeProductionTable.
3. Extract the start symbol. See makeStartSymbol.
4. Compile a short production table. See compileShortPT.
5. Return the grammar.

Value

A grammar object (list) with the attributes

- name (the filename of the grammar),
- ST (symbol table),
- PT (production table),
- Start (the start symbol of the grammar), and
- SPT (the short production table).

References

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

Examples

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

`compileShortPT` *Produces a production table with non-recursive productions only.*

Description

`compileShortPT()` produces a "short" production table from a context-free grammar. The short production table does not contain recursive production rules. Warning: No error checking implemented.

Usage

```
compileShortPT(G)
```

Arguments

`G`	A grammar with symbol table `ST`, production table `PT`, and start symbol `Start`.

Details

`compileShortPT()` starts with production rules whose right-hand side contains only terminals. It incrementally builds up the new PT until at least one production rule sequence from a non-terminal to a terminal symbol.

The short production rule provides for each non-terminal symbol a minimal finite derivation into terminals. Instead of the full production table, it is used for generating depth-bounded derivation trees.

Value

A (short) production table is a named list with 2 columns. The first column (the left-hand side `LHS`) is a vector of non-terminal identifiers. The second column (the right-hand side `RHS`) is a vector of vectors of numerical identifiers. `LHS[i]` derives into `RHS[i]`.

Examples

```
g<-compileBNF(booleanGrammar())
compileShortPT(g)
```

derive — *Derives the identifier list which expands the non-terminal identifier.*

Description

`derives()` returns the identifier list which expands a non-terminal identifier. Warning: No error checking implemented.

Usage

```
derive(RuleIndex, RHS)
```

Arguments

`RuleIndex`	An index (integer) in the production table.
`RHS`	The right-hand side of the production table.

Value

A vector of numerical identifiers.

See Also

Other Utility Functions: `id2symb()`, `isNonTerminal()`, `isTerminal()`, `rules()`, `symb2id()`

Examples

```
a<-booleanGrammar()$BNF
ST<-makeSymbolTable(a)
PT<-makeProductionTable(a,ST)
derive(1, PT$RHS)
derive(2, PT$RHS)
derive(3, PT$RHS)
derive(5, PT$RHS)
```

id2symb — *Convert a numeric identifier to a symbol.*

Description

`id2symb()` converts a numeric id to a symbol.

Usage

```
id2symb(Id, ST)
```

Arguments

Id	A numeric identifier (integer).
ST	A symbol table.

Value

- A symbol string if the identifier exists or
- an empty character string (`character(0)`) if the identifier does not exist.

See Also

Other Utility Functions: `derive()`, `isNonTerminal()`, `isTerminal()`, `rules()`, `symb2id()`

Examples

```
g<-compileBNF(booleanGrammar())
id2symb(1, g$ST)
id2symb(2, g$ST)
id2symb(5, g$ST)
id2symb(12, g$ST)
id2symb(15, g$ST)
identical(id2symb(15, g$ST), character(0))
```

`isNonTerminal` *Is the numeric identifier a non-terminal symbol?*

Description

`isNonTerminal()` tests if the numeric identifier is a non-terminal symbol.

Usage

```
isNonTerminal(Id, ST)
```

Arguments

Id	A numeric identifier (integer).
ST	A symbol table.

Details

`isNonTerminal()` is one of the most frequently used functions of a grammar-based genetic programming algorithm. Careful coding pays off! Do not index the symbol table as a matrix (e.g. `ST[2,2]`), because this is really slow!

Value

- TRUE if the numeric identifier is a terminal symbol.
- FALSE if the numeric identifier is a non-terminal symbol.
- NA if the symbol does not exist.

See Also

Other Utility Functions: derive(), id2symb(), isTerminal(), rules(), symb2id()

Examples

```
g<-compileBNF(booleanGrammar())
isNonTerminal(1, g$ST)
isNonTerminal(2, g$ST)
isNonTerminal(5, g$ST)
isNonTerminal(12, g$ST)
isNonTerminal(15, g$ST)
identical(isNonTerminal(15, g$ST), NA)
```

---

isTerminal *Is the numeric identifier a terminal symbol?*

Description

isTerminal() tests if the numeric identifier is a terminal symbol.

Usage

```
isTerminal(Id, ST)
```

Arguments

Id	A numeric identifier (integer).
ST	A symbol table.

Details

isTerminal() is one of the most frequently used functions of a grammar-based genetic programming algorithm. Careful coding pays off! Do not index the symbol table as a matrix (e.g. ST[2,2]), because this is really slow!

Value

- TRUE if the numeric identifier is a terminal symbol.
- FALSE if the numeric identifier is a non-terminal symbol.
- NA if the symbol does not exist.

See Also

Other Utility Functions: `derive()`, `id2symb()`, `isNonTerminal()`, `rules()`, `symb2id()`

Examples

```
g<-compileBNF(booleanGrammar())
isTerminal(1, g$ST)
isTerminal(2, g$ST)
isTerminal(5, g$ST)
isTerminal(12, g$ST)
isTerminal(15, g$ST)
identical(isTerminal(15, g$ST), NA)
```

`makeProductionTable` *Produces a production table.*

Description

`makeProductionTable()` produces a production table from a specification of a BNF. Warning: No error checking implemented.

Usage

```
makeProductionTable(BNF, ST)
```

Arguments

`BNF`	A character string with the BNF.
`ST`	A symbol table.

Value

A production table is a named list with elements `$LHS` and `$RHS`:

- The left-hand side `LHS` of non-terminal identifiers.
- The right-hand side `RHS` is represented as a vector of vectors of numerical identifiers.

The non-terminal identifier `LHS[i]` derives into `RHS[i]`.

Examples

```
a<-booleanGrammar()$BNF
ST<-makeSymbolTable(a)
makeProductionTable(a,ST)
```

makeRule *Transforms a single BNF rule into a production table.*

Description

makeRule() transforms a single BNF rule into a production table.

Usage

```
makeRule(Rule, ST)
```

Arguments

Rule	A rule.
ST	A symbol table.

Details

Because a single BNF rule can provide a set of substitutions, more than one line in a production table may result. The number of substitutions corresponds to the number of lines in the production table.

Value

A named list with 2 elements, namely $LHS and $RHS. The left-hand side $LHS is a vector of non-terminal identifiers and the right-hand side $RHS is a vector of vectors of numerical identifiers. The list represents the substitution of $LHS[i] by the identifier list $RHS[[i]].

Examples

```
c<-booleanGrammar()$BNF
ST<-makeSymbolTable(c)
c<-booleanGrammar()$BNF
b<-strsplit(c,";")[[1]]
a<-b[2:4]
a<-gsub(pattern=";",replacement="", paste(a[1], a[2], a[3], sep=""))
makeRule(a, ST)
```

makeStartSymbol *Extracts the numerical identifier of the start symbol of the grammar.*

Description

`makeStartSymbol()` returns the start symbol's numerical identifier from a specification of a context-free grammar in BNF. Warning: No error checking implemented.

Usage

```
makeStartSymbol(BNF, ST)
```

Arguments

BNF	A character string with the BNF.
ST	A symbol table.

Value

The numerical identifier of the start symbol of the BNF.

Examples

```
a<-booleanGrammar()$BNF
ST<-makeSymbolTable(a)
makeStartSymbol(a,ST)
```

makeSymbolTable *Build a symbol table from a character string which contains a BNF.*

Description

`makeSymbolTable()` extracts all terminal and non-terminal symbols from a BNF and builds a data frame with the columns Symbols (string), NonTerminal (0 or 1), and SymbolId (int). The symbol "NotExpanded" is added which codes depth violations of a derivation tree.

Usage

```
makeSymbolTable(BNF)
```

Arguments

BNF	A character string with the BNF.

Value

A data frame with the columns `Symbols`, `NonTerminal`, and `SymbolID`.

Examples

```
makeSymbolTable(booleanGrammar()$BNF)
```

---

newBNF *Convert grammar file into a constant function.*

Description

`newBNF()` reads a text file and returns a constant function which returns the BNF as a character string.

Usage

```
newBNF(filename, eol = "\n")
```

Arguments

`filename`	A file name.
`eol`	End-of-line symbol(s). Default: `"\n"`

Details

The purpose of this function is to include examples of grammars in packages.

Value

Returns a constant function which returns a BNF.

See Also

Other File I/O: `readBNF()`, `writeBNF()`

Examples

```
g<-booleanGrammar()
fn<-tempfile()
writeBNF(g, fn)
g1<-newBNF(fn)
unlink(fn)
```

readBNF *Read text file.*

Description

`readBNF()` reads a text file and returns a character string.

Usage

```
readBNF(filename, eol = "")
```

Arguments

`filename`	A file name.
`eol`	End-of-line symbol(s). Default: ""

Value

A named list with

- $filename the filename.
- $BNF a character string with the newline symbol \n.

See Also

Other File I/O: `newBNF()`, `writeBNF()`

Examples

```
g<-booleanGrammar()
fn<-tempfile()
writeBNF(g, fn)
g1<-readBNF(fn)
unlink(fn)
```

rules *Returns all indices of rules applicable for a non-terminal identifier.*

Description

`rules()` finds all applicable production rules for a non-terminal identifier.

Usage

```
rules(Id, LHS)
```

Arguments

Id	A numerical identifier.
LHS	The left-hand side of a production table.

Value

- A vector of indices of all applicable rules in the production table or
- an empty integer (integer(0)), if the numerical identifier is not found in the left-hand side of the production table.

See Also

Other Utility Functions: derive(), id2symb(), isNonTerminal(), isTerminal(), symb2id()

Examples

```
a<-booleanGrammar()$BNF
ST<-makeSymbolTable(a)
PT<-makeProductionTable(a,ST)
rules(5, PT$LHS)
rules(8, PT$LHS)
rules(9, PT$LHS)
rules(1, PT$LHS)
```

---

symb2id *Convert a symbol to a numeric identifier.*

Description

symb2id() converts a symbol to a numeric id.

Usage

```
symb2id(sym, ST)
```

Arguments

sym	A character string with the symbol, e.g. <fe> or "NOT".
ST	A symbol table.

Value

- A positive integer if the symbol exists or
- an empty integer (integer(0)) if the symbol does not exist.

See Also

Other Utility Functions: derive(), id2symb(), isNonTerminal(), isTerminal(), rules()

Examples

```
g<-compileBNF(booleanGrammar())
symb2id("<fe>", g$ST)
symb2id("NOT", g$ST)
symb2id("<fe", g$ST)
symb2id("NO", g$ST)
identical(symb2id("NO", g$ST), integer(0))
```

---

writeBNF *Write BNF into text file.*

Description

writeBNF() writes a character string into a textfile.

Usage

```
writeBNF(g, fn = NULL, eol = "\n")
```

Arguments

g	A named list with $filename and $BNF as a character string.
fn	A file name. Default: NULL.
eol	End-of-line symbol(s). Default: "\n"

Details

The user writes the BNF to a text file which he edits. The newline symbols are inserted after each substitution variant and after each production rule to improve the readability of the grammar by the user.

Value

Invisible NULL.

See Also

Other File I/O: newBNF(), readBNF()

Examples

```
g<-booleanGrammar()
fn<-tempfile()
writeBNF(g, fn)
g1<-readBNF(fn, eol="\n")
unlink(fn)
```

---

xegaBNF *Package xegaBNF*

Description

xegaBNF implements a grammar compiler for context-free languages specified in BNF and a few utility functions. The grammar compiler generates a grammar object. This object used by the package xegaDerivationTrees, as well as for grammar-based genetic programming (xegaGpGene) and grammatical evolution (xegaGeGene.

BNF (Backus-Naur Form)

Grammars of context-free languages are represented in Backus-Naur Form (BNF). See e.g. Backus et al. (1962).

The BNF is a meta-language for specifying the syntax of context-free languages. The BNF provides

1. non-terminal symbols,
2. terminal symbols, and
3. meta-symbols of the BNF.

A non-terminal symbol has the following form: <pattern>, where pattern is an arbitrary sequence of letters, numbers, and symbols.

A terminal symbol has the following form: "pattern", where pattern is an arbitrary sequence of letters, numbers, and symbols.

The BNF has three meta symbols, namely ::=, |, and ; which are used for the specification of production (substitution) rules. ::= separates the left-hand side of the rule from the right-hand side of the rule. ; indicates the end of a production rule. | separates the symbol sequences of a compound production rule. A production rule has the following form:

LHS ::= RHS;

where LHS is a single non-terminal symbol and RHS is either a simple symbol sequence or a compound symbol sequence.

A production rule with a simple symbol sequence specifies the substitution of the non-terminal symbol on the LHS by the symbol sequence of the RHS.

A production rule with a compound symbol sequence specifies the substitution of the non-terminal symbol on the LHS by one of the symbol sequences of the RHS.

Editing BNFs

The BNF may be stored in ASCII text files and edited with standard editors.

The Internal Representation of a Grammar Object

A grammar object is represented as a named list:

- $name contains the filename of the BNF.
- $ST the symbol table.
- $PT the production table.
- $Start the start symbol of the grammar.
- $SPT a short production table without recursive rules.

The Compilation Process

The main steps of the compilation process are:

1. Store the filename.
2. Make the symbol table. See `makeSymbolTable`.
3. Make the production table. See `makeProductionTable`.
4. Extract the start symbol. See `makeStartSymbol`.
5. Compile a short production table. See `compileShortPT`.
6. Return the grammar.

The User-Interface of the Compiler

`compileBNF(g)` where g is a character string with a BNF.

Utility Functions for xegaX-Packages

- isTerminal, isNonTerminal: For testing the symbol type of identifiers in a grammar object.
- rules, derives: For choosing rules and for substitutions.

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 population statistics dependent adaptive mechanisms and parallelization.
- The gene layer is split into 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 as e.g. 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.
- `xegaDerivationTrees` implements an abstract data type for derivation trees.

Copyright

License

MIT

URL

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

Installation

From CRAN by `install.packages('xegaBNF')`

Author(s)

Andreas Geyer-Schulz

References

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, Peter, Perlis, A. J., Ruthishauser, H., and Samelson, K. (1962) Revised Report on the Algorithmic Language ALGOL 60, IFIP, Rome.

Index