Package 'xegaBNF'

May 8, 2026

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

Version 1.0.0.5

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>) A preprocessor for macros which expand to standard BNF is included. The grammar compiler is an extension of 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 7.3.2

Suggests testthat (>= 3.0.0)

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2025-04-17 16:20:02 UTC

Contents

allTerminal *Are all symbols of vector of symbols terminal symbols?*

Description

Are all symbols of vector of symbols terminal symbols?

Usage

```
allTerminal(symbols, ST = ST)
```

Arguments

symbols	A vector of symbol identifiers.
ST	A symbol table.

Value

Boolean.

See Also

Other Compilation of short production table: cL(), directRecursion(), expandGrid(), expandRules(), findNextRuleForExpansion(), finiteRulesOfG(), nonTerminalsOfG(), smallestRules()

Examples

```
g<-compileBNF(booleanGrammar())
s<-c(2, 10, 3)
allTerminal(s, g$ST)
```

bindKvariables *R-code to bind variable names with values from a vector.*

Description

R-code to bind variable names with values from a vector.

Usage

```
bindKvariables(varSym, vecSym, k)
```

Arguments

varSym	String. Variable name (character part).
vecSym	String. Vector name.
k	Integer. Number of variables.

Details

Compiles R code for the assignment of a vector to single variables. The variable names are formed by catenating the variable name varSym to the character string of an integer in 1:k.

Value

R-code which assigns the content of the vector to a list of k variables with synthetic names.

Examples

```
bindKvariables("B", "v", 5)
```

---

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()
```

booleanGrammarK — *A constant function which returns the BNF (Backus-Naur Form) of a context-free grammar for the XOR problem with k boolean variables.*

Description

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

Usage

```
booleanGrammarK()
```

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

```
booleanGrammarK()
```

cL — *Combines two lists.*

Description

Combines two lists.

Usage

```
cL(x, y)
```

Arguments

x	A list.
y	A list.

Details

Each element of the result has the form `unlist(c(x[i], y[j]))` for all `i` and for all `j`.

Value

A list.

See Also

Other Compilation of short production table: `allTerminal()`, `directRecursion()`, `expandGrid()`, `expandRules()`, `findNextRuleForExpansion()`, `finiteRulesOfG()`, `nonTerminalsOfG()`, `smallestRules()`

Examples

```
a<-cL(c(1, 2), c(2, 3))
b<-cL(a, c(6))
```

`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 in the new PT at least one production rule exists for each non-terminal which replaces the non-terminal symbol by a list of terminal symbols.

The short production rule table provides for each non-terminal symbol a minimal finite derivation into terminals. It contains a finite subset of the context-free language as defined by the grammar `G`. Instead of the full production table, it is used for generating depth-bounded derivation trees. The first idea of defining such a finite part of the language is due to M. P. Schützenberger (1966).

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]`.

References

Schützenberger, M. P. (1966): Classification of Chomsky Languages. In: Steel, T. B. Jr. (Ed.) Formal Language Description Languages for Computer Programming. Proceedings of the IFIP Workshop on Formal Language Description Languages. North-Holland, Amsterdam, 100 -104.

See Also

Other Compiler Steps: `makeProductionTable()`, `makeStartSymbol()`, `makeSymbolTable()`

Examples

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

---

dataframePT *The dataframe of a production table of a grammar (readable).*

Description

The dataframe of a production table of a grammar (readable).

Usage

```
dataframePT(PT, G)
```

Arguments

PT	A production table of the grammar G.
G	A grammar object

Value

A dataframe of the production table.

See Also

Other Diagnostics: `printPT()`

Examples

```
g<-compileBNF(booleanGrammar())
cat("Production table:\n")
l<-dataframePT(g$PT, g)
cat("Short Production table:\n")
dataframePT(g$SPT, 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)
```

directRecursion — *Which production rules contain a direct recursion?*

Description

Which production rules contain a direct recursion?

Usage

```
directRecursion(G)
```

Arguments

G	A compiled context-free grammar.

Details

Direct recursion means the nonterminal on the LHS is also a symbol on RHS of production rule.

Value

Vector of booleans.

See Also

Other Compilation of short production table: allTerminal(), cL(), expandGrid(), expandRules(), findNextRuleForExpansion(), finiteRulesOfG(), nonTerminalsOfG(), smallestRules()

Examples

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

---

evenMacro *Is the number macro patterns even?*

Description

evenMacro tests if the macro start/end patterns are balanced (occur in even numbers).

Usage

```
evenMacro(BNFfn)
```

Arguments

BNFfn	A constant function which returns a BNF.

Value

Boolean.

See Also

Other Grammar Preprocessor: existsMacro(), pastePart(), preBNF()

Examples

```
evenMacro(booleanGrammar)
evenMacro(booleanGrammarK)
```

existsMacro *Does the grammar contain macros?*

Description

Macros (R-code) in grammars starts and ends with the pattern " //R//". existsMacro() tests if macro patterns are present in a grammar file.

Usage

```
existsMacro(BNFfn)
```

Arguments

BNFfn	A constant function which returns a BNF.

Value

Boolean.

See Also

Other Grammar Preprocessor: evenMacro(), pastePart(), preBNF()

Examples

```
existsMacro(booleanGrammar)
existsMacro(booleanGrammarK)
```

expandGrid *Expands a vector of symbol vectors.*

Description

Expands a vector of symbol vectors.

Usage

```
expandGrid(obj)
```

Arguments

obj	A vector of symbol vectors.

Value

A list of symbol vectors which is the Cartesian product of all symbol vectors in `obj`.

See Also

Other Compilation of short production table: `allTerminal()`, `cL()`, `directRecursion()`, `expandRules()`, `findNextRuleForExpansion()`, `finiteRulesOfG()`, `nonTerminalsOfG()`, `smallestRules()`

Examples

```
l<-list()
l[[1]]<-c(1, 2, 3)
l[[2]]<-c(4, 5)
expandGrid(l)
```

`expandRules` *Replaces rules with fNTs and terminals by a new set of rules with terminals.*

Description

Replaces rules with fNTs and terminals by a new set of rules with terminals.

Usage

```
expandRules(rPT, SPT, G)
```

Arguments

`rPT`	Rules fNTs and terminals.
`SPT`	Current short production table (SPT).
`G`	The grammar.

Value

The extended short production table.

See Also

Other Compilation of short production table: `allTerminal()`, `cL()`, `directRecursion()`, `expandGrid()`, `findNextRuleForExpansion()`, `finiteRulesOfG()`, `nonTerminalsOfG()`, `smallestRules()`

Examples

```
g<-compileBNF(booleanGrammar())
finiteRules<-finiteRulesOfG(g)
SPT<-newPT(LHS=g$PT$LHS[finiteRules], RHS=g$PT$RHS[finiteRules])
rest<-!(finiteRulesOfG(g) | directRecursion(g))
restPT<-newPT(LHS=g$PT$LHS[rest], RHS=g$PT$RHS[rest])
nSPT<-expandRules(rPT=restPT, SPT=SPT, g)
printPT(nSPT, g)
```

`findNextRuleForExpansion`

Find next rule which must be expanded.

Description

Find next rule which must be expanded.

Usage

```
findNextRuleForExpansion(PT, fNTS, G)
```

Arguments

`PT`	Production table.
`fNTS`	List of finite non terminals.
`G`	A grammar.

Value

A list of indices.

See Also

Other Compilation of short production table: `allTerminal()`, `cL()`, `directRecursion()`, `expandGrid()`, `expandRules()`, `finiteRulesOfG()`, `nonTerminalsOfG()`, `smallestRules()`

Examples

```
g<-compileBNF(booleanGrammar())
finiteRules<-finiteRulesOfG(g)
SPT<-newPT(LHS=g$PT$LHS[finiteRules], RHS=g$PT$RHS[finiteRules])
finiteNTs<-unique(SPT$LHS)
rest<-!(finiteRulesOfG(g) | directRecursion(g))
restPT<-newPT(LHS=g$PT$LHS[rest], RHS=g$PT$RHS[rest])
findNextRuleForExpansion(restPT, finiteNTs, g)
```

finiteRulesOfG *Which production rules produce only terminal symbols?*

Description

Which production rules produce only terminal symbols?

Usage

```
finiteRulesOfG(G)
```

Arguments

G	A compiled context-free grammar.

Value

Vector of booleans.

See Also

Other Compilation of short production table: allTerminal(), cL(), directRecursion(), expandGrid(), expandRules(), findNextRuleForExpansion(), nonTerminalsOfG(), smallestRules()

Examples

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

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]`.

See Also

Other Compiler Steps: `compileShortPT()`, `makeStartSymbol()`, `makeSymbolTable()`

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.

See Also

Other Compiler Steps: `compileShortPT()`, `makeProductionTable()`, `makeSymbolTable()`

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.

See Also

Other Compiler Steps: compileShortPT(), makeProductionTable(), makeStartSymbol()

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)
```

---

newPT *Constructs a new production table.*

Description

Constructs a new production table.

Usage

```
newPT(LHS, RHS)
```

Arguments

LHS	The vector of non-terminal identifiers.
RHS	A list of vectors of symbols.

Value

A production table (a named list) with the elements

1. $LHS: A vector of nonterminals.
2. $RHS: A list of vectors of symbols.

Examples

```
g<-compileBNF(booleanGrammar())
nPT<-newPT(g$PT$LHS, g$PT$RHS)
```

nonTerminalsOfG *Returns the list of symbol identifiers of nonterminal symbols in G.*

Description

Returns the list of symbol identifiers of nonterminal symbols in G.

Usage

```
nonTerminalsOfG(G)
```

Arguments

G	A compiled context-free grammar.

Value

The list of the symbol identifiers of all nonterminal symbols of grammar G

See Also

Other Compilation of short production table: allTerminal(), cL(), directRecursion(), expandGrid(), expandRules(), findNextRuleForExpansion(), finiteRulesOfG(), smallestRules()

Examples

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

pastePart *Catenates a vector of strings into a single string.*

Description

The vector elements are separated by a white space.

Usage

```
pastePart(tvec)
```

Arguments

tvec	A vector of strings.

Value

A string.

See Also

Other Grammar Preprocessor: `evenMacro()`, `existsMacro()`, `preBNF()`

Examples

```
a<-c("text", "text2")
pastePart(a)
```

`preBNF` *BNF preprocessing.*

Description

The BNF preprocessor executes macros (R-code embedded in a BNF grammar definition) and replaces the macros by the output they produce.

Usage

```
preBNF(BNFfn, genv = NULL)
```

Arguments

`BNFfn`	A constant function which returns a BNF.
`genv`	The list of bindings needed by the macros in the R-code.

Details

The embedded R-code starts with `" //R//"` and ends with `"" //R//"`. The preprocessor accepts a binding list which binds R objects their values. The macros are evaluated in an environment with these bindings. The output of each macro is inserted into the grammar file. It is expected that after preprocessing, the grammar file is in the BNF-notation. For example, generic grammar files can be provided for which the number of symbols of a certain type (e.g. variables) can be specified by the bindings.

Value

A list with elements `filename` and `BNF`

See Also

Other Grammar Preprocessor: `evenMacro()`, `existsMacro()`, `pastePart()`

Examples

```
a<-preBNF(booleanGrammar)
b<-preBNF(booleanGrammarK, list(k=5))
```

`printPT` *Print a production table of a grammar.*

Description

Print a production table of a grammar.

Usage

```
printPT(PT, G, verbose = TRUE)
```

Arguments

`PT`	A production table of the grammar G.
`G`	A grammar object
`verbose`	Print production table? Default: `TRUE`.

Value

An invisible list of the production table.

See Also

Other Diagnostics: `dataframePT()`

Examples

```
g<-compileBNF(booleanGrammar())
cat("Production table:\n")
l<-printPT(g$PT, g, verbose=TRUE)
cat("Short Production table:\n")
printPT(g$SPT, g, verbose=TRUE)
```

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)
```

---

smallestRules *List of rules with the smallest number of nonterminals.*

Description

List of rules with the smallest number of nonterminals.

Usage

```
smallestRules(PT)
```

Arguments

PT	Production table.

Value

List of indices of the production rule(s) with the smallest number of non terminals.

See Also

Other Compilation of short production table: `allTerminal()`, `cL()`, `directRecursion()`, `expandGrid()`, `expandRules()`, `findNextRuleForExpansion()`, `finiteRulesOfG()`, `nonTerminalsOfG()`

Examples

```
g<-compileBNF(booleanGrammar())
smallestRules(g$PT)
```

`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))
```

variableNamesBNF *Generate synthetic variable names as list of rules in BNF.*

Description

Generate synthetic variable names as list of rules in BNF.

Usage

```
variableNamesBNF(varNT, varSym, k)
```

Arguments

varNT	String. The non-terminal symbol for variables.
varSym	String. The variable names (character part).
k	Integer. Number of variables.

Details

Compiles BNF rules for variable names. The integers in `1:k` are the integer part of the variable names generated.

Value

Text vector with production rules for k variables.

See Also

Other Syntactic support.: `variableNamesLHS()`

Examples

```
cat(variableNamesBNF("<f0>", "D", 7))
```

variableNamesLHS *Generate synthetic variable names as list of rules in BNF.*

Description

Generate synthetic variable names as list of rules in BNF.

Usage

```
variableNamesLHS(varSym, k)
```

Arguments

varSym	String. The variable names (character part).
k	Integer. Number of variables.

Details

Compiles the LHS part of a BNF rule for variable names. The integers in 1:k are the integer part of the variable names generated.

Value

Text vector with production rules for k variables.

See Also

Other Syntactic support.: variableNamesBNF()

Examples

```
cat(variableNamesLHS("D", 7))
```

---

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.

URL

https://github.com/ageyerschulz/xegaBNF

Copyright

License

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.

See Also

Useful links:

- https://github.com/ageyerschulz/xegaBNF

Index