Package 'rly'

May 8, 2022

Type Package

Title Tools to Create Formal Language Parser

Version 1.7.4

Date 2022-05-08

Author Marek Jagielski [aut, cre, cph],
David M. Beazley [aut, cph],
Yasutaka Tanaka [ctb],
Henrico Witvliet [ctb]

Maintainer Marek Jagielski <marek.jagielski@gmail.com>

Description R implementation of the common parsing tools 'lex' and 'yacc'.

License MIT + file LICENSE

URL https://github.com/systemincloud/rly

BugReports https://github.com/systemincloud/rly/issues

Suggests testthat, knitr, rmarkdown

Encoding UTF-8

Depends R (>= 3.3.0)

Imports R6, futile.logger

RoxygenNote 6.1.1

Collate 'logger.R' 'lex.R' 'yacc.R' 'rly-package.R'

VignetteBuilder knitr

NeedsCompilation yes

Repository CRAN

Date/Publication 2022-05-08 13:00:02 UTC

R topics documented:

`lex` *Build a lexer*

Description

Build all of the regular expression rules from definitions in the supplied module

Usage

```
lex(module = NA, args = list(), debug = FALSE, debuglog = NA,
  errorlog = NA)
```

Arguments

`module`	R6 class containing lex rules
`args`	list of arguments that should be passed to constructor
`debug`	on and off debug mode
`debuglog`	custom logger for debug messages
`errorlog`	custom logger for error messages

Value

Lexer ready to use

Examples

```
TOKENS = c('NAME', 'NUMBER')
LITERALS = c('=','+','-','*','/', '(',')')

Lexer <- R6::R6Class("Lexer",
  public = list(
    tokens = TOKENS,
    literals = LITERALS,
    t_NAME = '[a-zA-Z_][a-zA-Z0-9_]*',
    t_NUMBER = function(re='\\d+', t) {
      t$value <- strtoi(t$value)
      return(t)
    },
    t_ignore = " \t",
    t_newline = function(re='\\n+', t) {
       t$lexer$lineno <- t$lexer$lineno + nchar(t$value)
```

```
        return(NULL)
      },
      t_error = function(t) {
        cat(sprintf("Illegal character '%s'", t$value[1]))
        t$lexer$skip(1)
        return(t)
      }
    )
)

lexer  <- rly::lex(Lexer)
lexer$input("5 + 3")
print(lexer$token()$value)
# [1] 5
print(lexer$token()$value)
# [1] "+"
print(lexer$token()$value)
# [1] 3
```

`Lexer` *Lexing Engine*

Description

The following Lexer class implements the lexer runtime. There are only a few public methods and attributes:

- input() - Store a new string in the lexer
- token() - Get the next token
- clone() - Clone the lexer
- lineno - Current line number
- lexpos - Current position in the input string

Usage

```
Lexer
```

Format

An `R6Class` generator object

LexToken *Lex Token*

Description

Token class. This class is used to represent the tokens produced

Usage

```
LexToken
```

Format

An R6Class generator object

LRParser *The LR Parsing engine*

Description

The LR Parsing engine

Usage

```
LRParser
```

Format

An R6Class generator object

NullLogger *Null logger is used when no output should be generated.*

Description

Does nothing.

Usage

```
NullLogger
```

Format

A R6Class object

Examples

```
debuglog <- NullLogger$new()
debuglog$info('This will not print')
```

RlyLogger *Print log message to file or console.*

Description

This object is a stand-in for a logging object created by the logging module. RLY will use this by default to create things such as the parser.out file. If a user wants more detailed information, they can create their own logging object and pass it into RLY. '

Usage

```
RlyLogger
```

Format

A R6Class object

Examples

```
debuglog <- rly::RlyLogger$new(".", "file.out")
debuglog$info('This is info message')

file.remove("file.out")
```

---

yacc *Build a parser*

Description

This function is entry point to the library

Usage

```
yacc(module = NA, args = list(), method = "LALR", debug = FALSE,
  start = NA, check_recursion = TRUE, debugfile = "parser.out",
  outputdir = NA, debuglog = NA, errorlog = NA)
```

Arguments

module	R6 class containing rules
args	list of arguments that should be passed to constructor
method	type of algorithm
debug	on and off debug mode
start	provide custom start method
check_recursion	should yacc look for recursions in rules
debugfile	the name of the custom debug output logs
outputdir	the dierectory of custom debug logs
debuglog	custom logger for debug messages
errorlog	custom logger for error messages

Value

Parser ready to use

Examples

```
TOKENS = c('NAME', 'NUMBER')
LITERALS = c('=','+','-','*','/', '(',')')

Parser <- R6::R6Class("Parser",
  public = list(
    tokens = TOKENS,
    literals = LITERALS,
    # Parsing rules
    precedence = list(c('left','+','-'),
                      c('left','*','/'),
                      c('right','UMINUS')),
    # dictionary of names
    names = new.env(hash=TRUE),
    p_statement_assign = function(doc='statement : NAME "=" expression', p) {
      self$names[[as.character(p$get(2))]] <- p$get(4)
    },
    p_statement_expr = function(doc='statement : expression', p) {
      cat(p$get(2))
      cat('\n')
    },
    p_expression_binop = function(doc="expression : expression '+' expression
                                                  | expression '-' expression
                                                  | expression '*' expression
                                                  | expression '/' expression", p) {
      if(p$get(3) == '+') p$set(1, p$get(2) + p$get(4))
      else if(p$get(3) == '-') p$set(1, p$get(2) - p$get(4))
      else if(p$get(3) == '*') p$set(1, p$get(2) * p$get(4))
      else if(p$get(3) == '/') p$set(1, p$get(2) / p$get(4))
    },
```

```
    p_expression_uminus = function(doc="expression : '-' expression %prec UMINUS", p) {
      p$set(1, -p$get(3))
    },
    p_expression_group = function(doc="expression : '(' expression ')'", p) {
      p$set(1, p$get(3))
    },
    p_expression_number = function(doc='expression : NUMBER', p) {
      p$set(1, p$get(2))
    },
    p_expression_name = function(doc='expression : NAME', p) {
      p$set(1, self$names[[as.character(p$get(2))]])
    },
    p_error = function(p) {
      if(is.null(p)) cat("Syntax error at EOF")
      else           cat(sprintf("Syntax error at '%s'", p$value))
    }
  )
)

parser <- rly::yacc(Parser)
```

YaccProduction *Object sent to grammar rule*

Description

This class is a wrapper around the objects actually passed to each grammar rule. Index lookup and assignment actually assign the .value attribute of the underlying YaccSymbol object. The lineno() method returns the line number of a given item (or 0 if not defined). The linespan() method returns a tuple of (startline,endline) representing the range of lines for a symbol. The lexspan() method returns a tuple (lexpos,endlexpos) representing the range of positional information for a symbol.

Usage

```
YaccProduction
```

Format

An R6Class generator object

Index