syntax

package standard library

Version: go1.20.1 Latest | Published: Feb 14, 2023 | License: BSD-3-Clause | Imports: 5 | Imported by: 899

Details ✓ Valid go.mod file ✓ Redistributable license ✓ Tagged version ✓ Stable version

Learn more

Repository cs.opensource.google/go/go

Links Report a Vulnerability

Documentation

Overview

Syntax

Package syntax parses regular expressions into parse trees and compiles parse trees into programs. Most clients of regular expressions will use the facilities of package regexp (such as Compile and Match) instead of this package.

Syntax

The regular expression syntax understood by this package when parsing with the Perl flag is as follows. Parts of the syntax can be disabled by passing alternate flags to Parse.

Single characters:

```
.              any character, possibly including newline (flag s=true)
[xyz]          character class
[^xyz]         negated character class
\d             Perl character class
\D             negated Perl character class
[[:alpha:]]    ASCII character class
[[:^alpha:]]   negated ASCII character class
\pN            Unicode character class (one-letter name)
\p{Greek}      Unicode character class
\PN            negated Unicode character class (one-letter name)
\P{Greek}      negated Unicode character class
```

Composites:

```
xy              x followed by y
x|y             x or y (prefer x)
```

Repetitions:

```
x*              zero or more x, prefer more
x+              one or more x, prefer more
x?              zero or one x, prefer one
x{n,m}          n or n+1 or ... or m x, prefer more
x{n,}           n or more x, prefer more
x{n}            exactly n x
x*?             zero or more x, prefer fewer
x+?             one or more x, prefer fewer
x??             zero or one x, prefer zero
x{n,m}?         n or n+1 or ... or m x, prefer fewer
x{n,}?          n or more x, prefer fewer
x{n}?           exactly n x
```

Implementation restriction: The counting forms x{n,m}, x{n,}, and x{n} reject forms that create a minimum or maximum repetition count above 1000. Unlimited repetitions are not subject to this restriction.

Grouping:

```
(re)            numbered capturing group (submatch)
(?P<name>re)    named & numbered capturing group (submatch)
(?:re)          non-capturing group
(?flags)        set flags within current group; non-capturing
(?flags:re)     set flags during re; non-capturing

Flag syntax is xyz (set) or -xyz (clear) or xy-z (set xy, clear z). The flags are:

i               case-insensitive (default false)
m               multi-line mode: ^ and $ match begin/end line in addition to begin/end t
s               let . match \n (default false)
U               ungreedy: swap meaning of x* and x*?, x+ and x+?, etc (default false)
```

Empty strings:

```
^               at beginning of text or line (flag m=true)
$               at end of text (like \z not \Z) or line (flag m=true)
\A              at beginning of text
\b              at ASCII word boundary (\w on one side and \W, \A, or \z on the other)
\B              not at ASCII word boundary
\z              at end of text
```

Escape sequences:

```
\a             bell (== \007)
\f             form feed (== \014)
\t             horizontal tab (== \011)
\n             newline (== \012)
\r             carriage return (== \015)
\v             vertical tab character (== \013)
\*             literal *, for any punctuation character *
\123           octal character code (up to three digits)
\x7F           hex character code (exactly two digits)
\x{10FFFF}     hex character code
\Q...\E        literal text ... even if ... has punctuation
```

Character class elements:

```
x              single character
A-Z            character range (inclusive)
\d             Perl character class
[:foo:]        ASCII character class foo
\p{Foo}        Unicode character class Foo
\pF            Unicode character class F (one-letter name)
```

Named character classes as character class elements:

```
[\d]           digits (== \d)
[^\d]          not digits (== \D)
[\D]           not digits (== \D)
[^\D]          not not digits (== \d)
[[:name:]]     named ASCII class inside character class (== [:name:])
[^[:name:]]    named ASCII class inside negated character class (== [:^name:])
[\p{Name}]     named Unicode property inside character class (== \p{Name})
[^\p{Name}]    named Unicode property inside negated character class (== \P{Name})
```

Perl character classes (all ASCII-only):

```
\d             digits (== [0-9])
\D             not digits (== [^0-9])
\s             whitespace (== [\t\n\f\r ])
\S             not whitespace (== [^\t\n\f\r ])
\w             word characters (== [0-9A-Za-z_])
\W             not word characters (== [^0-9A-Za-z_])
```

ASCII character classes:

```
[[:alnum:]]    alphanumeric (== [0-9A-Za-z])
[[:alpha:]]    alphabetic (== [A-Za-z])
[[:ascii:]]    ASCII (== [\x00-\x7F])
[[:blank:]]    blank (== [\t ])
[[:cntrl:]]    control (== [\x00-\x1F\x7F])
[[:digit:]]    digits (== [0-9])
[[:graph:]]    graphical (== [!-~] == [A-Za-z0-9!"#$%&'()*+,\-./:;<=>?@[\\\]^_`{|}~])
```

```
[[:lower:]]    lower case (== [a-z])
[[:print:]]    printable (== [ -~] == [ [:graph:]])
[[:punct:]]    punctuation (== [!-/:-@[-`{-~])
[[:space:]]    whitespace (== [\t\n\v\f\r ])
[[:upper:]]    upper case (== [A-Z])
[[:word:]]     word characters (== [0-9A-Za-z_])
[[:xdigit:]]   hex digit (== [0-9A-Fa-f])
```

Unicode character classes are those in unicode.Categories and unicode.Scripts.

Index

func IsWordChar(r rune) bool

type EmptyOp

    func EmptyOpContext(r1, r2 rune) EmptyOp

type Error

    func (e *Error) Error() string

type ErrorCode

    func (e ErrorCode) String() string

type Flags

type Inst

    func (i *Inst) MatchEmptyWidth(before rune, after rune) bool

    func (i *Inst) MatchRune(r rune) bool

    func (i *Inst) MatchRunePos(r rune) int

    func (i *Inst) String() string

type InstOp

    func (i InstOp) String() string

type Op

    func (i Op) String() string

type Prog

    func Compile(re *Regexp) (*Prog, error)

    func (p *Prog) Prefix() (prefix string, complete bool)

    func (p *Prog) StartCond() EmptyOp

    func (p *Prog) String() string

type Regexp

    func Parse(s string, flags Flags) (*Regexp, error)

    func (re *Regexp) CapNames() []string

    func (x *Regexp) Equal(y *Regexp) bool

    func (re *Regexp) MaxCap() int

    func (re *Regexp) Simplify() *Regexp

    func (re *Regexp) String() string

Constants

This section is empty.

Variables

This section is empty.

Functions

func IsWordChar

```
func IsWordChar(r rune) bool
```

IsWordChar reports whether r is considered a “word character” during the evaluation of the \b and \B zero-width assertions. These assertions are ASCII-only: the word characters are [A-Za-z0-9_].

Types

type EmptyOp

```
type EmptyOp uint8
```

An EmptyOp specifies a kind or mixture of zero-width assertions.

```
const (
    EmptyBeginLine EmptyOp = 1 << iota
    EmptyEndLine
    EmptyBeginText
    EmptyEndText
    EmptyWordBoundary
    EmptyNoWordBoundary
)
```

func EmptyOpContext

```
func EmptyOpContext(r1, r2 rune) EmptyOp
```

EmptyOpContext returns the zero-width assertions satisfied at the position between the runes r1 and r2. Passing r1 == -1 indicates that the position is at the beginning of the text. Passing r2 == -1 indicates that the position is at the end of the text.

type Error

```
type Error struct {
    Code ErrorCode
    Expr string
}
```

An Error describes a failure to parse a regular expression and gives the offending expression.

func (*Error) Error

```
func (e *Error) Error() string
```

type ErrorCode

```
type ErrorCode string
```

An ErrorCode describes a failure to parse a regular expression.

```
const (
    // Unexpected error
    ErrInternalError ErrorCode = "regexp/syntax: internal error"

    // Parse errors
    ErrInvalidCharClass      ErrorCode = "invalid character class"
    ErrInvalidCharRange      ErrorCode = "invalid character class range"
    ErrInvalidEscape         ErrorCode = "invalid escape sequence"
    ErrInvalidNamedCapture   ErrorCode = "invalid named capture"
    ErrInvalidPerlOp         ErrorCode = "invalid or unsupported Perl syntax"
    ErrInvalidRepeatOp       ErrorCode = "invalid nested repetition operator"
    ErrInvalidRepeatSize     ErrorCode = "invalid repeat count"
    ErrInvalidUTF8           ErrorCode = "invalid UTF-8"
    ErrMissingBracket        ErrorCode = "missing closing ]"
    ErrMissingParen          ErrorCode = "missing closing )"
    ErrMissingRepeatArgument ErrorCode = "missing argument to repetition operator"
    ErrTrailingBackslash     ErrorCode = "trailing backslash at end of expression"
    ErrUnexpectedParen       ErrorCode = "unexpected )"
    ErrNestingDepth          ErrorCode = "expression nests too deeply"
    ErrLarge                 ErrorCode = "expression too large"
)
```

func (ErrorCode) String

```
func (e ErrorCode) String() string
```

type Flags

```
type Flags uint16
```

Flags control the behavior of the parser and record information about regexp context.

```
const (
    FoldCase      Flags = 1 << iota // case-insensitive match
    Literal                         // treat pattern as literal string
    ClassNL                         // allow character classes like [^a-z] and [[:space
    DotNL                           // allow . to match newline
    OneLine                         // treat ^ and $ as only matching at beginning and
    NonGreedy                       // make repetition operators default to non-greedy
    PerlX                           // allow Perl extensions
    UnicodeGroups                   // allow \p{Han}, \P{Han} for Unicode group and neg
    WasDollar                       // regexp OpEndText was $, not \z
    Simple                          // regexp contains no counted repetition

    MatchNL = ClassNL | DotNL

    Perl        = ClassNL | OneLine | PerlX | UnicodeGroups // as close to Perl as poss
    POSIX Flags = 0                                         // POSIX syntax
)
```

type Inst

```
type Inst struct {
    Op   InstOp
    Out  uint32 // all but InstMatch, InstFail
    Arg  uint32 // InstAlt, InstAltMatch, InstCapture, InstEmptyWidth
    Rune []rune
}
```

An Inst is a single instruction in a regular expression program.

func (*Inst) MatchEmptyWidth

```
func (i *Inst) MatchEmptyWidth(before rune, after rune) bool
```

MatchEmptyWidth reports whether the instruction matches an empty string between the runes before and after. It should only be called when i.Op == InstEmptyWidth.

func (*Inst) MatchRune

```
func (i *Inst) MatchRune(r rune) bool
```

MatchRune reports whether the instruction matches (and consumes) r. It should only be called when i.Op == InstRune.

func (*Inst) MatchRunePos

added in go1.3

```
func (i *Inst) MatchRunePos(r rune) int
```

MatchRunePos checks whether the instruction matches (and consumes) r. If so, MatchRunePos returns the index of the matching rune pair (or, when len(i.Rune) == 1, rune singleton). If not, MatchRunePos returns -1. MatchRunePos should only be called when i.Op == InstRune.

func (*Inst) String

```
func (i *Inst) String() string
```

type InstOp

```
type InstOp uint8
```

An InstOp is an instruction opcode.

```
const (
    InstAlt InstOp = iota
    InstAltMatch
    InstCapture
    InstEmptyWidth
    InstMatch
    InstFail
    InstNop
    InstRune
    InstRune1
    InstRuneAny
    InstRuneAnyNotNL
)
```

func (InstOp) String

added in go1.3

```
func (i InstOp) String() string
```

type Op

```
type Op uint8
```

An Op is a single regular expression operator.

```
const (
    OpNoMatch        Op = 1 + iota // matches no strings
    OpEmptyMatch                   // matches empty string
    OpLiteral                      // matches Runes sequence
    OpCharClass                    // matches Runes interpreted as range pair list
    OpAnyCharNotNL                 // matches any character except newline
    OpAnyChar                      // matches any character
    OpBeginLine                    // matches empty string at beginning of line
    OpEndLine                      // matches empty string at end of line
    OpBeginText                    // matches empty string at beginning of text
    OpEndText                      // matches empty string at end of text
    OpWordBoundary                 // matches word boundary `\b`
    OpNoWordBoundary               // matches word non-boundary `\B`
    OpCapture                      // capturing subexpression with index Cap, optional 
    OpStar                         // matches Sub[0] zero or more times
    OpPlus                         // matches Sub[0] one or more times
    OpQuest                        // matches Sub[0] zero or one times
    OpRepeat                       // matches Sub[0] at least Min times, at most Max (M
    OpConcat                       // matches concatenation of Subs
    OpAlternate                    // matches alternation of Subs
)
```

func (Op) String

added in go1.11

```
func (i Op) String() string
```

type Prog

```
type Prog struct {
	Inst   []Inst
	Start  int // index of start instruction
	NumCap int // number of InstCapture insts in re
}
```

A Prog is a compiled regular expression program.

func Compile

```
func Compile(re *Regexp) (*Prog, error)
```

Compile compiles the regexp into a program to be executed. The regexp should have been simplified already (returned from re.Simplify).

func (*Prog) Prefix

```
func (p *Prog) Prefix() (prefix string, complete bool)
```

Prefix returns a literal string that all matches for the regexp must start with. Complete is true if the prefix is the entire match.

func (*Prog) StartCond

```
func (p *Prog) StartCond() EmptyOp
```

StartCond returns the leading empty-width conditions that must be true in any match. It returns ^EmptyOp(0) if no matches are possible.

func (*Prog) String

```
func (p *Prog) String() string
```

type Regexp

```
type Regexp struct {
	Op       Op // operator
	Flags    Flags
	Sub      []*Regexp  // subexpressions, if any
	Sub0     [1]*Regexp // storage for short Sub
	Rune     []rune     // matched runes, for OpLiteral, OpCharClass
	Rune0    [2]rune    // storage for short Rune
```

```
    Min, Max int        // min, max for OpRepeat
    Cap      int        // capturing index, for OpCapture
    Name     string     // capturing name, for OpCapture
}
```

A Regexp is a node in a regular expression syntax tree.

func Parse

```
func Parse(s string, flags Flags) (*Regexp, error)
```

Parse parses a regular expression string s, controlled by the specified Flags, and returns a regular expression parse tree. The syntax is described in the top-level comment.

func (*Regexp) CapNames

```
func (re *Regexp) CapNames() []string
```

CapNames walks the regexp to find the names of capturing groups.

func (*Regexp) Equal

```
func (x *Regexp) Equal(y *Regexp) bool
```

Equal reports whether x and y have identical structure.

func (*Regexp) MaxCap

```
func (re *Regexp) MaxCap() int
```

MaxCap walks the regexp to find the maximum capture index.

func (*Regexp) Simplify

```
func (re *Regexp) Simplify() *Regexp
```

Simplify returns a regexp equivalent to re but without counted repetitions and with various other simplifications, such as rewriting /(?:a+)+/ to /a+/. The resulting regexp will execute correctly but its string representation will not produce the same parse tree, because capturing parentheses may have been duplicated or removed. For example, the simplified form for /(x){1,2}/ is /(x)(x)?/ but both parentheses capture as $1. The returned regexp may share structure with or be the original.

func (*Regexp) String

```
func (re *Regexp) String() string
```

Source Files

View all

compile.go	parse.go	regexp.go
doc.go	perl_groups.go	simplify.go
op_string.go	prog.go	