# Expressions régulières

**Objectifs** : dans ce chapitre, vous apprendrez à utiliser et à créer des expressions régulières pour la manipulation des chaînes.

**Sommaire :** Un programme français… en anglais – de l’anglais au français – encore plus loin : de l’anglais au choix de la langue

**Ressources** : les programmes de test sont présents dans le sous-répertoire t*raduction* du répertoire e*xemples*.

## Introduction

Les expressions régulières sont un outil puissant pour la détection et la manipulation de chaînes de caractères. Vous allez découvrir la syntaxe des expressions régulières et apprendre à utiliser les fonctions de l'unité *RegExpr* livrée avec le compilateur Free Pascal.

Avant de commencer, faites un premier essai de compilation. Voici un programme qui affiche le numéro de version de l'unité *RegExpr* :

|  |
| --- |
| uses  sysutils, regexpr;  const  S = 'Expressions r'#130'guli'#138'res version %d.%d';  begin  writeLn(Format(S, [TRegExpr.VersionMajor(), TRegExpr.VersionMinor()])); |

Vous obtenez le résultat suivant :

|  |
| --- |
| Expressions régulières version 0.952 |

La documentation et les exemples originellement joints à l'unité n'ont pas été inclus dans le paquetage de Lazarus. Il est toutefois possible de les télécharger sur le site personnel de l'auteur, Andrey V. Sorokin : <http://regexpstudio.com/TRegExpr/TRegExpr.html>

La syntaxe supportée par l'unité *RegExpr* est un sous-ensemble des expressions régulières de Perl.

### Définition des expressions régulières

On appelle *expression régulière* une chaîne de caractères qui représente un ensemble de chaînes de caractères.

Ainsi, la chaîne 'abc', considérée comme expression régulière, représente un ensemble contenant un seul élément, la chaîne 'abc'.

### La fonction ExecRegExpr() de l'unité RegExpr

La fonction *ExecRegExpr()* de l'unité *RegExpr* permet de savoir si une chaîne donnée appartient à l'ensemble représenté par une expression :

|  |
| --- |
| uses  regexpr;  begin  writeLn(ExecRegExpr('abc', 'abc')); // TRUE |

Pour être plus exact, la fonction vérifie qu'une partie au moins de la chaîne passée en second argument appartient à l'ensemble représenté par l'expression régulière passée en premier argument :

|  |
| --- |
| writeLn(ExecRegExpr('abc', 'abcd')); // TRUE |

### Les caractères symbolisant le début et la fin de la chaîne

Si vous voulez vous assurer que la chaîne entière, et non pas seulement l'une de ses parties, appartient à l'ensemble représenté par l'expression régulière, il faut ajouter au début et à la fin de celle-ci deux caractères qui signifient respectivement « début de la chaîne » et « fin de la chaîne » :

writeLn(ExecRegExpr('^abc$', 'abc')); // TRUE

writeLn(ExecRegExpr('^abc$', 'abcd')); // FALSE

### Échappement des caractères spéciaux

Et si vous voulez représenter l'ensemble contenant comme seul élément la chaîne '^abc$' ?

Vous devez adjoindre aux caractères spéciaux '^' et '$' un autre caractère signifiant que les caractères en question sont à interpréter, pour cette fois, de façon littérale. C'est ce qu'on appelle, d'après l'anglais, « échapper » ces caractères spéciaux.

En l'occurrence, il s'agit du caractère '\' :

|  |
| --- |
| writeLn(ExecRegExpr('\^abc\$', '^abc$')); // TRUE |

Le caractère '\' peut d'ailleurs servir à « s'échapper » lui-même :

|  |
| --- |
| writeLn(ExecRegExpr('\\', '\')); // TRUE |

## Cases de l'échiquier

Il est temps de découvrir le vrai pouvoir des expressions régulières. Vous allez maintenant composer et utiliser des expressions représentant des ensembles contenant plus d'un élément.

### Comparaison avec le type ensemble du langage Pascal

Supposez par exemple que vous vouliez savoir si une chaîne donnée est le nom d'une case de l'échiquier. Certes il est possible de le faire en Pascal, sans passer par les expressions régulières :

|  |
| --- |
| {$B-}  function IsChessSquare(aStr: string): boolean;  const  E1: set of char = ['a'..'h'];  E2: set of char = ['1'..'8'];  begin  Result := (Length(aStr) = 2)  and (aStr[1] in E1)  and (aStr[2] in E2);  end;  var  s: string;  begin  s := 'e2';  writeLn(IsChessSquare(s)); // TRUE |

Remarquez que pratiquement, la directive {$B-} n'est pas nécessaire, car elle ne fait que confirmer la valeur par défaut. Elle est utilisée ici plutôt comme un commentaire, pour signaler que la fonction *IsChessSquare* n'est pas conçue pour supporter l'évaluation longue des expressions booléennes. Voyez-vous pourquoi ?[[1]](#footnote-1)

### Classes de caractères

Voici l'expression régulière, ou plutôt *une* expression régulière, qui représente l'ensemble des cases de l'échiquier :

|  |
| --- |
| '[a-h][1-8]' |

Soit dit en passant, cette expression aurait pu aussi s'écrire de la façon suivante :

|  |
| --- |
| '[abcdefgh][12345678]' |

Dans un cas comme dans l'autre, vous aurez défini deux classes de caractères (très semblables aux deux ensembles de votre première fonction), et vous demandez *un* caractère de l'une et *un* caractère de l'autre. Le nombre un est implicite : c'est la valeur par défaut.

Pour finir, ajoutez à votre expression les caractères signifiant « début et fin de la chaîne » :

|  |
| --- |
| '^[a-h][1-8]$' |

Voici la nouvelle version de la fonction *IsChessSquare()* :

|  |
| --- |
| uses  regexpr;  function IsChessSquareRE(aStr: string): boolean;  const  EXPR = '^[a-h][1-8]$';  begin  Result := ExecRegExpr(EXPR, aStr);  end; |

Vous commencez à entrevoir l'un des atouts de ce langage que sont les expressions régulières : la brièveté.

## Numéros de téléphone

### Classes prédéfinies

Imaginez que vous vouliez savoir si une chaîne de caractères donnée contient un numéro de téléphone, soit une suite de 10 chiffres. Il y a justement  une expression régulière pour dire cela : « une suite de dix chiffres ». À vrai dire, il y en a même plusieurs. Vous pourriez d'abord l'écrire ainsi :

|  |
| --- |
| '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]' |

Heureusement, il y a une classe prédéfinie qui correspond exactement à la classe '[0-9]'. On la note '\d'. Votre expression devient donc :

|  |
| --- |
| '\d\d\d\d\d\d\d\d\d\d' |

Le caractère '\', qui tout à l'heure servait à forcer l'interprétation littérale d'un caractère spécial, sert ici au contraire à donner une valeur spéciale au caractère 'd', la valeur « chiffre » (*digit* en anglais).

### Indication du nombre de caractères

La chaîne que vous voulez détecter étant composée de dix caractères de la même classe, vous pouvez présenter votre expression régulière de la façon suivante :

|  |
| --- |
| '\d{10}' |

Et vous l'utiliserez ainsi :

|  |
| --- |
| uses  regexpr;  begin  writeLn(ExecRegExpr('\d{10}', 'maison 0123456789')); // TRUE |

La fonction *ExecRegExpr()* renvoie *TRUE*, car la chaîne passée comme second argument contient une suite de dix chiffres.

### Caractères éventuels

Tout cela est bel et bon, direz-vous, mais quelquefois les numéros de téléphone contiennent des espaces. Et quelquefois, ils n'en contiennent pas.

Eh bien, il y a encore une expression régulière pour dire cela. Vous allez utiliser les caractères suivants :

|  |
| --- |
| '\s?' |

Ces trois caractères signifient, dans le langage des expressions régulières : « un espace éventuel ». Désormais votre fonction est capable de détecter aussi bien les numéros avec espaces que les numéros sans espaces :

|  |
| --- |
| const  EXPRESSION = '\d{2}\s?\d{2}\s?\d{2}\s?\d{2}\s?\d{2}';  begin  writeLn(ExecRegExpr(EXPRESSION, 'maison 01 23 45 67 89')); // TRUE  writeLn(ExecRegExpr(EXPRESSION, 'maison 0123456789')); // TRUE |

### Alternative

Très bien, très bien, sauf qu'avec ce système, même une chaîne espacée de façon irrégulière sera validée :

|  |
| --- |
| 'maison 01 23456789' |

Supposez qu'on ne veuille accepter que les chaînes correctement formatées, tout en conservant la possibilité de choisir entre le format avec espace et le format sans espace. Comment procéder ?

Il y a une expression régulière pour dire cela : « ou une suite de dix chiffres non espacés, ou des chiffres groupés par deux » :

|  |
| --- |
| '(\d{10}|\d{2}\s\d{2}\s\d{2}\s\d{2}\s\d{2})' |

C'est-à-dire :

|  |
| --- |
| '(' + '\d{10}' + '|' + '\d{2}\s\d{2}\s\d{2}\s\d{2}\s\d{2}' + ')' |

Le caractère | veut dire « ou ». Les parenthèses indiquent où se terminent les expressions alternatives.

## Dates

### Petite révision

Intéressez-vous à la chaîne que renvoie la fonction *DateToStr()* de l'unité *SysUtils*. En France, le format de cette chaîne est le suivant :

|  |
| --- |
| '00/00/0000' |

Vous savez déjà tout ce qu'il faut savoir pour trouver l'expression régulière représentant l'ensemble de toutes les chaînes respectant ce format. Voici cette expression :

|  |
| --- |
| '\d\d/\d\d/\d\d\d\d' |

Mais pas si vite ! Vous pouvez être un peu plus précis :

|  |
| --- |
| '[0-3]\d/[01]\d/[12]\d\d\d' |

Le lecteur du troisième millénaire aura une modification à faire. Laquelle ?

Voici le code complet :

|  |
| --- |
| uses  regexpr, sysutils;  const  EXPRESSION = '[0-3]\d/[01]\d/[12]\d\d\d';    var  s: string;  begin  s := DateToStr(Now);  writeLn(ExecRegExpr(EXPRESSION, s)); // TRUE  end. |

### La classe TRegExpr

Savoir qu'une chaîne donnée contient une date ou est une date, c'est fort bien, mais supposez que vous vouliez extraire cette date. Comment faire ?

Pour cela, la fonction *ExecRegExpr()* ne vous suffit plus. Vous devez faire connaissance avec la classe *TRegExpr* :

|  |
| --- |
| const  EXPRESSION = '[0-3]\d/[01]\d/[12]\d\d\d';  var  s: string;  e: TRegExpr;    begin  s := DateToStr(Now);    e := TRegExpr.Create;  e.Expression := EXPRESSION;  e.Exec(s);    writeLn(e.Match[0]); // 09/06/2015  writeLn(e.MatchPos[0]); // 1  writeLn(e.MatchLen[0]); // 10    e.Free;  end. |

Comme vous le voyez, les variables *Match*, *MatchPos* et *MatchLen* sont des tableaux, dont seul le premier élément vous intéresse pour le moment. Vous apprendrez plus loin à quoi servent les autres éléments de ces tableaux.

### La fonction ReplaceRegExpr()

Maintenant que vous avez une expression régulière représentant l'ensemble de toutes les dates possibles, vous pouvez vous en servir, non seulement pour détecter une date, mais (pourquoi pas) pour la remplacer. Pour ce faire, vous utiliserez la fonction *ReplaceRegExpr()* :

|  |
| --- |
| const  EXPRESSION = '([0-3]\d)/([01]\d)/([12]\d\d\d)';  var  s: string;  begin  s := '00/00/2000';  writeLn(ReplaceRegExpr(EXPRESSION, s, DateToStr(Now), FALSE)); // 09/06/2015 |

La valeur *FALSE* du quatrième paramètre signifie que vous souhaitez un remplacement simple, c'est-à-dire sans substitutions.

### Remplacement complexe par substitution

En vous servant toujours du même exemple, voyez maintenant ce que sont les substitutions.

Imaginez que vous vouliez conserver les nombres trouvés dans la chaîne, mais en changer l'ordre. Vous passerez la valeur *TRUE* comme quatrième paramètre, et dans votre chaîne de remplacement vous insérerez les séquences de caractères '$1', '$2' et '$3' :

|  |
| --- |
| writeLn(ReplaceRegExpr(EXPRESSION, s, '$3-$2-$1', TRUE)); // 2015/06/09 |

Auparavant, vous avez ajouté à votre expression régulière des parenthèses qui servent à délimiter les trois sous-expressions dont le résultat sera substitué, dans la chaîne de remplacement, aux séquences de caractères '$1', '$2' et '$3'.

### Remplacement complexe par fonction.

Mais la fonction *ReplaceRegExpr()* vous permet, au besoin, un contrôle encore plus grand sur ce processus de remplacement complexe. En effet, le troisième paramètre peut être une fonction, qui composera la chaîne de remplacement à partir des éléments capturés dans la chaîne originale.

Vous pouvez par exemple, non seulement changer l'ordre des parties de la date, comme vous l'avez déjà fait précédemment, mais aussi remplacer le mois par son nom en anglais.

Voici la fonction qui composera la chaîne de remplacement :

|  |
| --- |
| function TMyApplication.MyReplaceFunc(ARegExpr: TRegExpr): string;  const  MONTH\_NAMES: array[1..12] of string = (  'January',  'February',  'March',  'April',  'May',  'June',  'July',  'August',  'September',  'October',  'November',  'December'  );  var  i: Integer;  begin  with ARegExpr do  begin  i := StrToInt(Match[2]);  Result := MONTH\_NAMES[i] + ' ' + Match[1] + ', ' + Match[3]; // June 09, 2015  end;  end; |

La variable *Match[0]* vous aurait fourni la date entière ; les variables *Match[1]* et *Match[3]* vous donnent les parties de la date correspondant aux première et troisième sous-expressions.

La fonction *ReplaceRegExpr()* est appelée de la façon suivante :

|  |
| --- |
| var  e: TRegExpr;  begin  e := TRegExpr.Create;  e.Expression := '(\d{2})/(\d{2})/(\d{4})';  {$IFDEF OBJFPC}  writeLn(e.Replace(DateToStr(Now), @MyReplaceFunc));  {$ELSE}  writeLn(e.Replace(DateToStr(Now), MyReplaceFunc));  {$ENDIF}  e.Free; |

Il est à noter que la fonction qui compose la chaîne de remplacement doit être une méthode de la classe au sein de laquelle la fonction *ReplaceRegExpr()* est appelée.

L'exemple ci-dessus est basé sur le type *TCustomApplication*. Le projet a été créé en choisissant, dans le menu Projet/Nouveau projet, le modèle « Application console ». De cette façon nous n'avez plus qu'à ajouter la déclaration de votre fonction de remplacement dans la déclaration de la classe *TmyApplication* fournie par Lazarus :

|  |
| --- |
| type  TMyApplication = class(TCustomApplication)  protected  procedure DoRun; override;  public  constructor Create(TheOwner: TComponent); override;  destructor Destroy; override;  function MyReplaceFunc(ARegExpr: TRegExpr): string;  end; |

## Validation d'une chaîne FEN

Il est temps de récapituler vos connaissances sur la syntaxe des expressions régulières.

Pour ce faire, vous allez imaginer une expression qui servira à contrôler la validité d'une chaîne représentant une position au jeu des échecs.

### Brève présentation de la notation Forsyth-Edwards

La notation Forsyth-Edwards permet de faire tenir dans une chaîne de caractères toutes les données constituant une position au jeu des échecs : placement des pièces, couleur active, droit au roque, droit à la prise « en passant », nombre de demi-coups, nombre de coups.

La position de départ se note ainsi :

|  |
| --- |
| 'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1' |

Comme vous le voyez, la chaîne est faite de six sous-chaînes séparées par des espaces.

La première chose que vous allez faire est précisément de vérifier que ces sous-chaînes sont bien présentes au nombre de six. Pour cela, vous allez utiliser la fonction *SplitRegExpr(*).

### La fonction SplitRegExpr()

La fonction *SplitRegExpr()* reçoit trois arguments : l'expression régulière à remplacer, la chaîne à traiter et une liste de chaînes pour recueillir le résultat.

|  |
| --- |
| function TFENValidator.IsFEN(const aInputStr: string): boolean;  var  a: TStrings;  begin  a := TStringList.Create;  SplitRegExpr(' ', aInputStr, a);  result := (a.Count = 6); |

Si le résultat obtenu est conforme au résultat attendu, vous appelez une seconde fois la fonction *SplitRegExpr()*, pour obtenir les sous-chaînes correspondant aux différentes lignes de l'échiquier :

|  |
| --- |
| var  a, b: TStrings;  begin  a := TStringList.Create;  b := TStringList.Create;  SplitRegExpr(' ', aInputStr, a);  Result := (a.Count = 6);  if Result then  begin  SplitRegExpr('/', a[0], b);  Result := (b.Count = 8);  end; |

Si à ce stade la variable *Result* vaut toujours *TRUE*, vous pouvez passer à l'étape suivante, qui est de contrôler les différents champs un à un au moyen d'expressions régulières appropriées.

### Expressions utilisées

Voici les expressions dont vous vous servirez :

|  |
| --- |
| const  WHITEKING = 'K';  BLACKKING = 'k';  PIECES = '^[1-8BKNPQRbknpqr]+$';  ACTIVE = '^[wb]$';  CASTLING = '^[KQkq]+$|^\-$';  ENPASSANT = '^[a-h][36]$|^\-$';  HALFMOVE = '^\d+$';  FULLMOVE = '^[1-9]\d\*$'; |

Il n'y a rien dans ces expressions que nous n'ayons déjà rencontré, mis à part le caractère spécial '\*' qui signifie que le caractère en question (un chiffre) peut apparaître « autant de fois qu'on veut ou pas du tout ».

Vous n'avez plus qu'à vérifier que les chaînes contenues dans les listes a et b correspondent bien au motif, au moyen de la fonction *ExecRegExpr()*.

### Remplacement complexe

Les chaînes de la liste b vont toutefois être soumises à une vérification un peu plus sophistiquée. En effet, vous ne nous voulez pas seulement savoir si ces chaînes ne contiennent que des caractères autorisés : vous voulez également vous assurer que le nombre total de cases pour chaque ligne est bien de huit.

Pour ce faire, vous allez remplacer les chiffres représentant le nombre de cases vides successives par une quantité correspondante de symboles répétés. Ainsi la ligne 'pp1ppppp' deviendra 'pp-ppppp'. De cette façon vous connaîtrez le nombre total de cases, d'après la longueur de la chaîne.

Voici la fonction qui va effectuer ce remplacement :

|  |
| --- |
| function TFENValidator.ExpandDigit(aRegExpr: TRegExpr): string;  const  SYMBOL = '-';  begin  result := '';  with aRegExpr do  result := StringOfChar(SYMBOL, StrToInt(Match[0]));  end; |

### Exemple complet

Et voici l'exemple complet :

|  |
| --- |
| {$MODE DELPHI}  {$B-}  uses  classes, sysutils, regexpr;  type  TFENValidator = class  public  function ExpandDigit(aRegExpr: TRegExpr): string;  function IsFEN(const aInputStr: string): boolean;  end;  var  loglist: TStringList;  logfile: string;  procedure ToLog(const aText: string);  begin  loglist.Add(Format('%s %s', [TimeToStr(Now), aText]));  end;  function TFENValidator.ExpandDigit(aRegExpr: TRegExpr): string;  const  SYMBOL = '-';  begin  result := '';  with aRegExpr do  Result := StringOfChar(SYMBOL, StrToInt(Match[0]));  end;  function TFENValidator.IsFEN(const aInputStr: string): boolean;  const  WHITEKING = 'K';  BLACKKING = 'k';  PIECES = '^[1-8BKNPQRbknpqr]+$';  ACTIVE = '^[wb]$';  CASTLING = '^[KQkq]+$|^\-$';  ENPASSANT = '^[a-h][36]$|^\-$';  HALFMOVE = '^\d+$';  FULLMOVE = '^[1-9]\d\*$';  var  a, b: TStrings;  i: Integer;  e: TRegExpr;  s: string;  begin  a := TStringList.Create;  b := TStringList.Create;  e := TRegExpr.Create;  e.Expression := '\d';  SplitRegExpr(' ', aInputStr, a);  result := (a.Count = 6);  if Result then  begin  SplitRegExpr('/', a[0], b);  Result := (b.Count = 8);  end;  if result then  begin  Result := Result and ExecRegExpr(WHITEKING, a[0]);  Result := Result and ExecRegExpr(BLACKKING, a[0]);  for i := 0 to 7 do  begin  Result := Result and ExecRegExpr(PIECES, b[i]);  if Result then  begin  s := b[i];  repeat  s := e.Replace(s, ExpandDigit);  until not ExecRegExpr('\d', s);  ToLog(Format('%s %s', [{$I %LINE%}, s]));  Result := Result and (Length(s) = 8);  end;  end;  Result := Result and ExecRegExpr(ACTIVE, a[1]);  Result := Result and ExecRegExpr(CASTLING, a[2]);  Result := Result and ExecRegExpr(ENPASSANT, a[3]);  Result := Result and ExecRegExpr(HALFMOVE, a[4]);  Result := Result and ExecRegExpr(FULLMOVE, a[5]);  end;  a.Free;  b.Free;  e.Free;  end;  const  SAMPLE: array[1..5]of string = (  'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1',  'rnbqkbnr/pp1ppppp/8/2p5/4P3/8/PPPP1PPP/RNBQKBNR w KQkq c6 0 2',  'rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2',  '4k3/8/8/8/8/8/4P3/4K3 w - - 5 39',  '5k3/8/8/8/8/8/4P3/4K3 w - - 5 39'  );  var  i: Integer;  begin  loglist := TStringList.Create;  logfile := ChangeFileExt({$I %FILE%}, '.log');  with TFENValidator.Create do  for i := Low(SAMPLE) to High(SAMPLE) do  writeLn(BoolToStr(IsFEN(SAMPLE[i]), TRUE));  with loglist do  begin  SaveToFile(logfile);  Free;  end;  end. |

L'une des chaînes du tableau SAMPLE n'est pas valide. Voyez-vous laquelle ?

Si vous ne voyez pas, rendez-vous dans le fichier **\*.LOG** :

|  |
| --- |
| -----k--- |

Il y a une case de trop sur cette ligne : la chaîne n'est pas valide.

## Deux fonctions utiles

Vous avez fait le tour de la syntaxe des expressions régulières et des fonctions de l'unité *RegExpr*. Il vous reste à faire connaissance avec deux fonctions qui peuvent rendre service.

### La fonction QuoteRegExprMetaChars()

La fonction *QuoteRegExprMetaChars()* reçoit comme argument une expression régulière dont on aurait oublié « d'échapper » les caractères spéciaux. La fonction détecte ces caractères et les fait précéder du symbole '\'. Par exemple, le code suivant :

|  |
| --- |
| writeLn(QuoteRegExprMetaChars('(abc)')); |

donnera ce résultat :

|  |
| --- |
| \(abc\) |

Pratique, n'est-ce pas ?

### La fonction RegExprSubExpressions()

La fonction *RegExprSubExpressions()* extrait les sous-expressions d'une expression donnée :

|  |
| --- |
| var  liste: TStringList;  i: Integer;  begin  liste := TStringList.Create;    if RegExprSubExpressions('(\d)(\d)', liste, FALSE) = 0 then  for i := 0 to liste.Count - 1 do  writeLn(liste.Strings[i]); |

Lorsque l'extraction s'est faite sans erreur, la fonction renvoie 0. C'est le cas ici, et voici ce qu'affiche votre programme :

|  |
| --- |
| (\d)(\d)  \d  \d |

Le premier élément de la liste est l'expression complète ; les deux éléments suivants sont les sous-expressions.

Le troisième paramètre de la fonction *RegExpr()* permet d'autoriser ou non la syntaxe étendue. Concrètement, comme vous l'apprend la documentation de l'unité *RegExpr*, ce paramètre doit avoir la valeur *TRUE* si le modificateur m est activé.

Encore un mot pour vous dire ce que sont ces modificateurs, et vous aurez terminé.

### Modificateurs

Les modificateurs permettent de changer le comportement des fonctions de la bibliothèque. Il y a six modificateurs, désignés par les lettres i, m, s, g, x, r.

Le modificateur i, par exemple, permet de ne pas tenir compte de la différence entre majuscules et minuscules. Voici comment on l'utilise :

|  |
| --- |
| var  re: TRegExpr;  begin  re := TRegExpr.Create;  re.ModifierI := TRUE; |

Le modificateur m, dont nous avons déjà parlé, étend la signification des caractères spéciaux '^' et '$' aux débuts et aux fins de ligne à l'intérieur de la chaîne traitée.

## Conclusion

Il reste trois ou quatre petites choses dont nous aurions pu parler, à commencer par la fonction des quatre autres modificateurs, mais peut-être pensez-vous, avec raison, que vous savez déjà pratiquement tout ce qu'il faut savoir sur les expressions régulières et l'unité *RegExpr* d'Andrey V. Sorokin. À vous de jouer !

1. Avec {$B+}, une erreur pourrait survenir au cas où la chaîne en entrée ferait moins de deux caractères de long du fait du test des caractères 1 et 2 dans la suite de l’expression ! [↑](#footnote-ref-1)