# Regex et grammaires en Perl 6

Journées Perl

Paris, 9-10 juin 2017

Laurent Rosenfeld

# Les regexes de Perl 6

- Perl 6 a refondu les expressions régulières
  - Ne sont plus rétrocompatibles avec Perl 5
  - On peut utiliser la syntaxe Perl 5 avec l'adverbe :P5 ou :Perl5 :

```
next if m:P5/[aeiou]/;  # syntaxe P5
next if m/ <[aeiou]> /; # syntaxe P6
```

  - Objectifs : simplifier, rendre plus cohérent, droit d'inventaire sur héritage, permettre nouvelles constructions plus riches ou plus expressives

# Différences entre P5 et P6 (1)

- Utilise l'opérateur smart match **~~** (et **!~~**)

```
say "Reconnu" if "abcdef" ~~ / ab.d /;    # -> Affiche : Reconnu
```

  - Espace par défaut non significatif dans un motif
  - Les reconnaissances alimentent l'objet **$/**

```
say $/ if "abcdef" ~~ m{ ab . d };        # -> Affiche : 「abcd」
```

  - Le motif travaille par défaut sur la variable **$_**

```
for "abcdef" { say ~$/ if /a . ** 2 d/ }; # → abcd
```

  - Autres quantificateurs : **+, *, ?, +?, *?**

```
given "abcccdef" {say ~$/ if /ab \w**2..5 e/}; # -> abcccde
```

# Différences entre P5 et P6 (2)

- Classes de caractères
  - Métacaractères
    - Espace : \s  \h  \v  \t (et \S  \H  \V  \T)
    - Caractère numérique \d (et \D)
    - Caractère alphanumérique : \w (et \W)
  - Propriétés Unicode
    - Lettre <:L>, nombre <:N>, capitale <:Lu>

```
say $/ if "Perl" ~~ /<:Lu> <:!Lu> <:!N> ./;   # -> 「Perl」
```

    - Classes personnalisées: <[...]> et <-[...]>

```
say $/ if "abbbceabbabc" ~~ /<[a..c e]> +/;   # -> 「abbbceabbabc」
say $/ if "abbbceabba" ~~ /<[a..e] - [c]> +/; # -> 「abbb」
```

# Reconnaître une date

- Trouver une date au format JJ-MM-AAAA

```
my $jj = rx /\d\d/; my $mm = rx /\d\d/; my $an = rx /\d ** 4/;
say "Date = $/" if "Noël = 25-12-2017" ~~ /<$jj>'-'<$mm>'-'<$an>/;
```

  - Valider le mois

```
my $mm = rx { 1 <[0..2]>     # 10 to 12
            || 0 <[1..9]>    # 01 to 09
};
```

  - Validation du mois avec une assertion:

```
my $mm = rx / \d ** 2 <?{ 1 <= $/ <= 12 }> /;
```

  - Validation du jour:

```
my $jj = rx / \d ** 2 <?{ 1 <= $/ <= 31 }> /;
```

# Différences entre P5 et P6 (3)

- Alternatives
  - Première reconnaissance **||** et la plus longue **|**

```
say ~$/ if "abcdef" ~~ /ab || abcde/;       # -> ab
say ~$/ if "abcdef" ~~ /ab |  abcde/;       # -> abcde
```

- Regroupements et captures
  - Les crochets regroupent
  - Les parenthèses regroupent et capturent

```
say "Nombre = $0" if  "Nb 42" ~~ /'Nb ' (\d+) /;  # -> Nombre = 42
say "Nombre = $/[0]" if "Nb 42" ~~ /'Nb ' (\d+)/; # -> Nombre = 42
say "0: $0; 1: $1" if 'abc' ~~ /(a) b (c)/;       # -> 0: a; 1: c
say "$/[0] $/[0][1]" if 'abc' ~~ /a ((b)(c))/;    # -> bc c
```

# Les adverbes (ou modificateurs)

- Ignorer la casse (`:ignorecase` ou `:i`)

```
say ~$/ if "BAC" ~~ m:i/ ac /;          # -> AC
say ~$/ if "BAC" ~~ /:i a/;             # -> A
say ~$/ if "BAC" ~~ /a :i c/;           # -> () (échec)
my $regex = rx :i /a/;
say ~$0 if "BAC" ~~ /(<$regex>)/;       # → A
```

- Crochets et parenthèses limitent la portée

```
/ (:i a b) c /;          # reconnaît 'ABc' mais pas 'ABC'
/ [:i a b] c /;          # reconnaît 'ABc' mais pas 'ABC'
```

- Rendre les espaces significatifs (:sigspace)

```
say so "BAC" ~~ m:i:s/ a c /;           # -> False
say so "BAC" ~~ m:i:s/ ac /;            # ->True
```

- Interdire le retour arrière (`:ratchet` ou `:r`)

```
say 'abc' ~~ / \w+ . /;                 # -> ｢abc｣
say 'abc' ~~ / :r \w+ . /;              # -> Nil
```

# Captures nommées

- On peut donner un nom aux captures:

```
'abc' ~~ / $<nom_capture> = [ \w+ ] /;
say $<nom_capture>;                          # -> 「abc」
say ~$/{ "nom_capture"};                     # -> abc
```

  - Utilisation de $/ dans un contexte de hash

```
'decompte=23' ~~ / $<variable>=\w+ '=' $<valeur>=\w+ /;
say "$/<variable> => $/{"valeur"}";      # -> decompte => 23
for $/.kv -> $x, $y { say "$x => $y" };
# valeur => 23
# variable => decompte
```

  - Méthodes de la classe Match (sur l'objet $/)

```
say "abcde" ~~ / bc /;       # ->「bc」
say join " ", $/.orig, $/.from, $/.to;   # abcde 1 3
say join " ", $/.prematch, $/.postmatch; # -> a de
```

# Regex nommées

- On peut déclarer des regex nommées
  - Création : my regex nom { corps de la regex }
  - Syntaxe semblable à une fonction ou méthode
  - Crée une capture de même nom

```
my regex ligne { \N*\n }
if "abc\ndef" ~~ /<ligne> def/ {
    say "Première ligne: ", $<ligne>.chomp; # Première ligne: abc
}
```

- Les regex nommées peuvent être regroupées en grammaires
  - C'est souhaitable, c'est même leur raison d'être

# Règles nommées (1)

- Les *regex* nommées sont un exemple de règles nommées
- Il existe deux autres sortes de règles:
  - Les *tokens* ont un adverbe :ratchet implicite
  - Pas de retour arrière (pas de backtracking)
  - Les *rules* ont en plus un adverbe :sigspace
  - Espaces significatifs et pas de retour arrière
- On peut *appeler* une règle dans une autre:

```
my rule chiffres { \d+ }
my rule décimal { <chiffres> \. <chiffres> }
say "Cet objet coûte 13.45 euros" ~~ /<décimal>/; # -> 「13.45 」
```

# Règles nommées (2)

- Un *token* ne fait pas de retour arrière

```
my token lettres { abc .+ e };
say "abcde" ~~ /<lettres>/ ?? "Réussit" !! "Échoue"; # ->  Échoue

# Avec un quantificateur frugal (non gourmand), ça marche:
my token lettres { abc .+? e };
say "abcde" ~~ /<lettres>/ ?? "Réussit" !! "Échoue"; # ->  Réussit
```

- Mais ça ne marcherait pas avec une *rule*

```
my rule  lettres { abc .+? e };
say "abcde" ~~ /<lettres>/ ?? "Réussit" !! "Échoue"; # ->  Échoue

# En supprimant les espaces dans le motif, ça marche
my rule  lettres { abc.+?e };
say "abcde" ~~ /<lettres>/ ?? "Réussit" !! "Échoue"; # ->  Réussit
```

# Reconnaître une date, une IP

- Une date avec des règles

```
my rule jj { \d\d <?{ 1 <= $/ <= 31 }> }
my rule mm { \d\d <?{ 1 <= $/ <= 12 }> }
my regex an { \d ** 4 }
say"Date = $/" if "Noël = 25-12-2017" ~~ /<jj>'-'<mm>'-'<an>/;
```

- Une adresse IPv4

```
my token octet {(\d ** 1..3) <?{0 <= $0 <= 255 }> }
my token ip { <octet> ** 4 % '.' }
say "IP: $/" if "21.34.252.42" ~~ / <ip> /; # -> IP: 21.34.252.42
```

- Difficultés :
  - Ordre de déclaration des règles
  - Problème d'appels récursifs
  - Espace de noms
  - Réutilisabilité limitée

# Créer une grammaire

- Une grammaire regroupe des règles

```
grammar IPv4 {
    token TOP { <octet> ** 4 % '.' }
    token octet {(\d ** 1..3) <?{0 <= $0 <= 255 }> }
}
say $/ if IPv4.parse("232.44.56.34");
```

- Ce qui imprime:

```
「232.44.56.34」
 octet => 「232」
  0 => 「232」
 octet => 「44」
  0 => 「44」
 octet => 「56」
  0 => 「56」
 octet => 「34」
  0 => 「34」
```

# Dissection d'une grammaire

- Notre grammaire IPv4 :
  - Crée un espace de nom (plus de déclaration avec my)
  - N'est plus sensible à l'ordre des déclarations (*token* TOP déclaré avant *token* octet qu'il utilise)
  - Peut donc réaliser des appels récursifs de règles
  - TOP, point de départ par défaut de l'analyse grammaticale avec .parse
  - Hérite des méthodes comme .parse ou .parsefile
- Une grammaire est une classe, dont les règles nommées sont les méthodes

# Héritage de grammaires

- Grammaire parente Message :

```
grammar Message {
    rule texte     { <salutation> $<corps>=<ligne>+? <fin> }
    rule salutation    { [Salut|Bonjour] $<dest>=\S+? ',' }
    rule fin    { [à|'@'] ['+'|plus] ',' $<auteur>=.+ }
    token ligne    { \N* \n}
}
```

- Grammaire fille MessageFormel :

```
grammar MessageFormel is Message {
    rule salutation { [Cher|Chère] $<dest>=\S+? ',' }
    rule fin { Bien cordialement ',' $<auteur>=.+ }
}
```

- On peut créer une grammaire héritant de celle de Perl et modifiant certaines choses

# Classes et objets d'actions

- Une classe d'actions définit des actions à faire pour une règle qui réussit

```
grammar ArithmGrammar {
    token TOP { \s* <num> \s* <operation> \s* <num> \s*}
    token operation { <[*+/-]> }
    token num { \d+ | \d+\.\d+ | \.\d+ }
}
class ArithmActions {
    method TOP($/) {
        given $<operation> {
            when '*' { $/.make([*] $<num>) }
            when '+' { $/.make([+] $<num>) }
            when '/' { $/.make($<num>[0] / $<num>[1]) }
            when '-' { $/.make([-] $<num>) }
        }
    }
}
my $match = ArithmGrammar.parse("6 * 2.5", :actions(ArithmActions));
say $match.made;
```

# Une grammaire parsant du JSON

- JSON définit des objets (liste de paires clef/valeur) et des tableaux (valeurs CSV)

```
grammar JSON-Grammar {
    token TOP       { \s* [ <object> | <array> ] \s* }
    rule object     { '{' \s* <pairlist> '}' \s* }
    rule pairlist   {  <pair> * % \, }
    rule pair       {  <string>':' <value> }
    rule array      { '[' <valueList> ']'}
    rule valueList  {  <value> * % \, }
    token string    {  \" <-[\n " \t]>* \"  }
    token number    {
        [\+|\-]?
        [ \d+ [ \. \d+ ]? ] | [ \. \d+ ]
        [ <[eE]> [\+|\-]? \d+ ]?
    }
    token value     { <object> | <array> | <string> | <number>
                    | true   | false   | null
    }
}
```

# Une calculette avec précédence

- Respecter la précédence entre opérateurs
- Utilisation de parenthèses

```
grammar Calculator {
    rule TOP             { <expr> }
    rule expr            { <term> + % <plus-minus-op> }
    token plus-minus-op { [< + - >] }
    rule term            { <atom> + % <mult-div-op> }
    token mult-div-op    { [< * / >] }
    rule atom {
        | <num> { make +$<num> }
        | <paren-expr> { make $<paren-expr>.made}
    }
    rule num             { <sign> ? [\d+ | \d+\.\d+ | \.\d+ ] }
    rule paren-expr      { '(' <expr> ')' }
    token sign           { [< + - >] }
}
my $res = Calculator.parse("(((2+3)*(5-2))-1)*3", :actions(CalcActions));
```

# Conclusion

- En résumé, Perl 6 nous offre :
  - Des regex refondues, plus logiques, plus puissantes, plus expressives
  - Les grammaires permettent de parser des formats de texte moins structurés
  - Perl 6 est parsé avec une grammaire elle-même écrite en Perl 6
  - Les grammaires peuvent étendre le language (en héritant de la grammaire de Perl 6)

# Conclusion (2)

- Pour en savoir plus:
  - En français :
    - http : //laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/regex-grammaire-puissance/
  - En anglais :
    - http : //greenteapress.com/wp/think-perl-6/
    - https : //docs.perl6.org
- Questions ?

# La classe d'actions JSON

```
class JSON-actions {
    method TOP($/) {
        make $/.values.[0].made;
    };
    method object($/)    { make $<pairlist>.made.hash.item; }
    method pairlist($/)  { make $<pair>>>.made.flat; }
    method pair($/)      { make $<string>.made => $<value>.made; }
    method array($/)     { make $<valueList>.made.item; }
    method valueList($/) { make [$<value>.map(*.made)]; }
    method string($/)    { make ~$0 }
    method number($/)    { make +$/.Str; }
    method value($/) {
        given ~$/ {
            when "true"  {make Bool::True;}
            when "false" {make Bool::False;}
            when "null"  {make Any;}
            default { make $<val>.made;}
        }
    }
}
```

# Classe d'actions de la calculette

```
class CalcActions {
    method TOP ($/) { make $<expr>.made }
    method expr ($/)       { $.calculate($/, $<term>, $<plus-minus-op>) }
    method term ($/)       { $.calculate($/, $<atom>, $<mult-div-op>) }
    method paren-expr ($/) { make $<expr>.made; }
    method calculate ($/, $operands, $operators) {
        my $result = (shift $operands).made;
        while my $op = shift $operators {
            my $num = (shift $operands).made;
            given $op {
                when '+' { $result += $num; }
                when '-' { $result -= $num; }
                when '*' { $result *= $num; }
                when '/' { $result /= $num; }
                default  { die "unknown operator "}
            }
        }
        make $result;
    }
}
```

# Tests avec la calculette

```
for |< 3*4 5/6 3+5 74-32 5+7/3 5*3*2 (4*5) (3*2)+5 4+3-1/5 4+(3-1)/4 >,
    "12 + 6 * 5", " 7 + 12 + 23", " 2 + (10 * 4) ", "3 * (7 + 7)" {
    my $result = Calculator.parse($_, :actions(CalcActions));
    printf "%-15s %8.3f\n", $/,  $result.made if $result;
}
------------------------
3*4              12.000
5/6               0.833
3+5               8.000
74-32            42.000
5+7/3             7.333
5*3*2            30.000
(4*5)            20.000
(3*2)+5          11.000
4+3-1/5           6.800
4+(3-1)/4         4.500
12 + 6 * 5       42.000
 7 + 12 + 23     42.000
 2 + (10 * 4)    42.000
3 * (7 + 7)      42.000
```

# Tests avec calculette (2)

```
for "(((2+3)*(5-2))-1)*3", "2 * ((4-1)*((3*7) - (5+2)))"  {
    my $result = Calculator.parse($_, :actions(CalcActions));
    printf "%-30s %.3f\n", $/,  $result.made if $result;
}

-------------

(((2+3)*(5-2))-1)*3             42.000
2 * ((4-1)*((3*7) - (5+2)))     84.000
```