PCRE, czyli bogatsze wyrażenia regularne

PCRE, czyli *Perl Compatible Regular Expressions*, to niezależna implementacja wyrażeń regularnych, inspirowana wyrażeniami regularnymi w języku *Perl*. Pierwotnie PCRE dostępne były jako biblioteka w języku *C*. Obecnie wyrażeń tych możemy używać nawet z poleceniem `grep` (opcja `-P`) i tak będziemy ich używać podczas naszych zajęć.

Przewagę *PCRE* prezentujemy poprzez przykłady. Omawiając przykłady wprowadzimy składnię nowych funkcjonalności.

Przykład. Napisz wyrażenie regularne dopasowujące się jedynie do siedmioliterowych wyrazów w słowniku scrabblisty w których występuje spójny podciąg *myk*.

Używając standardowych wyrażeń regularnych można to zrobić tak:

```
grep -E '^(myk....|.myk...|..myk..|...myk.|....myk)$' scrabble.txt
```

Nie jest to w pełni satysfakcjonujące rozwiązanie i łatwo wyobrazić sobie zapytania w których długość wyrażenia do napisania osiąga rozmiar uniemożliwiający jego wygodne stosowanie. Oto alternatywne rozwiązanie używające `PCRE`:

```
grep -P '^(?=.{7}$).*myk.*' scrabble.txt
```

Powyższe wyrażenie regularne używa jednego z wariantów *lookarounds*. Są to polecenia, które możemy wstawić do wyrażenia regularnego, powodujące, że blok przed (lub po) poleceniu może być dopasowany tylko wtedy gdy warunek w poleceniu jest spełniony. Oto lista wszystkich lookarounds z przykładami. (Zdecydowaliśmy się nie tłumaczyć nazw lookarounds na polski.)

(i) *lookahead after the match*: `'\d+(?= złotych)'`
Przykład dopasowania: `100` w ciągu `100 złotych`

(ii) *lookahead before the match*: `'(?=\d+ złotych)\d+'`
Przykład dopasowania: `100` w ciągu `100 złotych`
Zwróć uwagę, że funkcja dwu powyższych wyrażeń jest taka sama ale *lookahead before the match* jest mniej efektywny.

(iii) *negative lookahead after the match*: `\d+(?! złotych)`
Przykład dopasowania: `100` w ciągu `100 hrywien`, ale nie w ciągu `100 złotych`. Zastanów się, czy w ciągu `100 złotych` coś zostanie dopasowane ?

(iv) *lookbehind before the match* `(?<=czarny )kot`
Przykład dopasowania: `kot` w ciągu `czarny kot`

(v) *negative lookbehind before the match* `(?<!czarny )kot`
Przykład dopasowania: `kot` w ciągu `mały kot`

Przykład. Napisz wyrażenie regularne dopasowujące się do linii zawierających dokładnie jedno wystąpienie ciągu `kosmos`.

Gdyby chodziło o dokładnie jedno wystąpienie, powiedzmy, litery `k` to można by to zrobić przy pomocy następującego wyrażenia:

```
^[^k]*k[^k]*$
```

Jednak jeśli chodzi o dokładne wystąpienie ciągu `kosmos`, następujące wyrażenie jest jedną z najprostszych opcji:

```
^(?=.*kosmos)(?!.*kosmos.*kosmos).*$
```

Przykład. Napisz wyrażenie regularne, które dopasowuje się do tekstu pomiędzy znacznikami `begin` i `end`. Szczegółowy opis które fragmenty chcemy dopasować znajduje się w zadaniu programistycznym E na satori.

Zacznijmy od bardzo naiwnego wyrażenia regularnego i spróbujmy je naprawiać krok po kroku.

```
begin.*end
```

Zauważmy najpierw, że wywołanie

```
echo "aaa begin end end aaa" | grep 'begin.*end'
```

dopasuje się do tekstu `begin end end`. Tymczasem w specyfikacji mamy, że blok wyróżniony (który chcemy dopasować) powinien być zamknięty pierwszym ciągiem `end` po rozpoczynającym ciągu `begin`. Błąd wynika z tego, że wyrażenia regularne są domyślnie interpretowane zachłannie. W PCRE możemy zmieniać to zachowanie.

```
begin.*?end
```

Znacznik `?` w sekwencji `.*?` interpretujemy, jako zmianę działania `.*` z dopasowywującego jak najdłuższy ciąg, na dopasowywujące jak najkrótszy ciąg pasujący do wyrażenia. Zatem interpreter w tym przypadku będzie szukać najkrótszego ciągu po którym następuje ciąg `end`.

Oto zwięzły zestaw przykładów operujący podobnymi znacznikami:

(i) `A++`, dopasowuje się zachłannie do jednego lub więcej A i sprawdza jedynie maksymalne dopasowanie nie próbując mniejszych; takie wyrażenie nazywamy *zaborczym*;
(ii) `A+`, dopasowuje się zachłannie do jednego lub więcej A, zaczyna sprawdzanie od maksymalnego dopasowanie ale w razie niepowodzenia sprawdza też kolejno mniejsze dopasowania;
(iii) `'A+.'` dopasuje się do `AAA` (ponieważ po dopasowaniu `AAA` nie ma kolejnego znaku, zostanie dopasowany ciąg `AA`, a ostatnie `A` potraktowane jako znak znajdujący się na końcu wzorca), natomiast `'A++.'` nie dopasuje się do `AAA` (nie zostanie sprawdzone dopasowanie mniejsze niż `AAA`).
(iv) znacznik `++` najczęściej jest używany w sytuacji gdy mamy pewność, że jeśli jest dopasowanie to powinno być maksymalne. Na przykład kiedy chcemy dopasować

liczbę, czyli ciąg cyfr i następujące po spacji słowo to efektywnie będzie użyć wyrażenia `[0-9]++ [a-z]++`;

(v) znacznik `++` przydatny jest również, gdy chcemy zapamiętać maksymalne dopasowanie jako grupę do której później się odwołamy. Patrz przykład poniżej.

(vi) `A*+` działa analogicznie zaborczo, pozwala jednak dopasować się do zero lub więcej liter `A`

Aby lepiej uzmysłowić działanie `++` przypomnijmy sobie przykład z *negative lookahead after the match*: `\d+(?! złotych)`. Wzorzec ten nie dopasuje `100` w ciągu `100 złotych`, ale dopasuje `10`, ponieważ nie zabraniamy aby po dopasowaniu wystepował ciąg `0 złotych`. Najprostszym i najbardziej eleganckim sposobem poprawienia tego wzorca jest użycie właśnie zaborczego modyfikatora: `\d++(?! złotych)`.

Wróćmy do przykładu ze znacznikami `begin` i `end`. Kolejnym zasadniczym problemem jest to, że obecne wyrażenie regularne nie wyszukuje bloków rozpiętych na więcej niż jednej linii. Tak naprawdę to wina zarówno polecenia grep, które przetwarza wejście linia po linii, niezależnie, jak i samego wyrażenia regularnego, w którym znak `.` nie dopasowuje się do znaku końca linii `\n`.

W związku z powyższym możemy poprawić dwie rzeczy:

(i) wywołać `grep` z opcją `-z`, dzięki czemu wejście dopasowywane będzie do wzorca w jednym przejściu, a nie linia po linii;

(ii) wyrażenie regularne powinno być poprzedzone modyfikatorem `(?s)`, dzięki czemu znak `.` będzie dopasowywać się także do znaku `\n`.

Zatem poprawione wywołanie polecenia to:

```
grep -Pz '(?s)begin.*?end'
```

Następnym problemem do rozwiązania jest to, że wzorzec dopasowuje się do bloków wyróżnionych wraz ze znacznikami ograniczającymi. Jeśli chcemy mieć dopasowanie jedynie do znaków wewnątrz bloku to powinniśmy skorzystać z *lookarounds*.

```
grep -Pz '(?s)(?<=begin).*?(?=end)'
```

Musimy jeszcze obsłużyć sytuację, zgodnie z treścią zadania na satori, że ostatni blok może nie mieć znacznika końca `end` jeśli skończy się wraz z końcem wejścia. Wykorzystamy do tego `\z` dopasowywujący znak końca pliku/wejścia. Ostateczny kształt polecenia to:

```
grep -Pz '(?s)(?<=begin).*?(?=(end|\z))'
```

Przykład. Na wejściu otrzymujemy linie zawierające ciągi binarne (jedyne znaki w linii to `0` i `1`). Mamy znaleźć wszystkie wystąpienia ciągu `11` takie że pierwsza jedynka jest na nieparzystym miejscu (pierwszy znak w linii jest na pierwszym miejscu).

Oto pierwsza próba rozwiązania:

```
grep -P '(..)*?11'
```

Wyrażenie dopasuje ciąg `11`, ale dopasowane zostaje także wszystko przed `11`. Dlatego kolejna próba rozwiązania będzie wyglądać tak:

```
grep -P '(..)*?\K11'
```

Komenda `\K` powoduje, że ciąg który się dopasował do danego momentu do wzorca jest wymazany z dopasowania.

Jak już wiemy, w wyrażeniach regularnych blok zaznaczony nawiasami okrągłymi zapamiętuje nam tekst dopasowany do bloku, do którego później możemy się odwołać poprzez `\1`,...,`\9`.

Okazuje się, że w PCRE blok możemy również traktować jako funkcję, którą możemy wywołać w wyrażeniu regularnym.

Przykład. Wyszukaj wszystkie palindromy w słowniku scrabblisty. Używając standardowych wyrażeń regularnych potrafiliśmy wyszukać palindromy określonej długości np. `'^(.)(.).\2\1$'` dopasowuje palindromy pięcioliterowe.

W PCRE możemy łatwo wyszukać wszystkie palindromy:

```
grep -P '^((.)(?1)?\2|.?)$'
```

Instrukcja `(?1)` powoduje wywołanie funkcji oznaczonej przez pierwszy blok. W tym przypadku jest to wywołanie rekurencyjne. Instrukcja `(?R)` powoduje wywołanie całego wyrażenia jako funkcji. Dlaczego następujące wyrażenie nie będzie działać dla palindromów?

```
grep -P '^(.)(?R)?\1|.?$'
```

Przykład. Wyszukaj linie postaci $a^n b^m a^n$ dla $1 \leq m \leq n$.

```
'^(?=(a+))a*(a(?2)?b)\1$'
```

Blokom możemy nadawać własne nazwy:

```
(?<slowo>[a-z]+) (?&slowo)
```

Powyżej zdefiniowaliśmy blok o nazwie słowo i wywołaliśmy go dalej w wyrażeniu jako funkcję. Powyższe wyrażenie dopasuje się np. do `krowa pies`.

Funkcje można także pre-definiować przed rozpoczęciem właściwego wyrażenia regularnego:

```
(?x) # włącza niewrażliwość wyrażenia na białe znaki
(?(DEFINE)  # początek bloku definiującego
# definicja funkcji ilosc
```

```
(?<ilosc>dużo|kilka|pięć)

# definicja funkcji jaki
(?<przymiotnik>niebieskich|dużych|interesujących)

# defincja funkcji obiekty
(?<obiekty>samochodów|słoni|problemów)

# definicja funkcji rzeczownik_fraza
(?<rzeczownik_fraza>(?&ilosc)\ (?&przymiotnik)\ (?&obiekty))

# definicja funkcji czynnosc
(?<czynnosc>pożycz|rozwiąż|poskładaj)
)          # koniec bloku definiującego

(?&rzeczownik_fraza)\ (?&czynnosc)\ (?&rzeczownik_fraza)
```

Niestety polecenie `grep -P` nie przyjmuje wzorców połamanych znakiem `\n`. To jest cały wzorzec musi być w jednej linii. Ponieważ pisanie złożonego wyrażenia z wieloma predefiniowanymi funkcjami jest nie do ogarnięcia w jednej linii proponujemy pisanie skryptu w pliku, a dopiero przed odpaleniem skryptu skasowanie wszystkich znaków złamania linii (albo prawie wszystkich, bo chcemy zachowac pierwszą linię skryptu `#!/bin/bash` nietkniętą). Oczywiście można to zrobić poleceniem `sed`, np. tak:

```
sed '1! {:a $! {N; ba;};
                s/\n//g;s/$/\n/}' < kecim-liczby.sh > kecim-liczby-out.sh
```