Wstęp do wyrażeń regularnych na platformie .NET

v1.1

Miłosz Orzeł

Celem artykułu jest zapoznanie Czytelnika z podstawowymi elementami składni wyrażeń regularnych oraz sposobami ich wykorzystania na platformie .NET. Postaram się zachęcić tych, którzy jeszcze nie mieli do czynienia z tym tematem do poznania tego potężnego narzędzia.

05.2007 (poprawki 07.2007)
Pierwsza wersja tego artykułu została nagrodzona w konkursie CodeGuru.pl.

Proszę o informacje jeśli zauważysz jakiś błąd w tekście artykułu lub dołączonym do niego kodzie.
ommail@wp.pl morzel.net

Czym są wyrażenia regularne?

Wyrażenia regularne (regex, RE) możemy traktować jako język, za pomocą którego da się precyzyjnie określić wzorzec występowania określonego ciągu znaków w łańcuchu tekstowym.

Wyrażenia regularne pozwalają na efektywną ocenę poprawności składniowej tekstu, jego przeszukiwanie oraz przetwarzanie...

Po co nam wyrażenia regularne?

Bardzo często pisane przez nas oprogramowanie musi w pewnym momencie wykonać operacje związane z walidacją lub parsowaniem tekstu. Może np. zajść konieczność sprawdzenia czy wpisany przez użytkownika identyfikator ma prawidłowy format albo wykonania sekwencji operacji w zależności od danych znajdujących się w pliku tekstowym o złożonej strukturze.

Chociaż wyrażenia regularne są obszernym zagadnieniem a ich składnia nie wygląda zbyt zachęcająco: `^[d-z]_\d{3}_(xyz|pki)_\d{2}_\w{2,7}.sql$`, na pewno warto poświęcić dzień nauki na opanowanie podstaw tego tematu. Istotne jest to, że elementarna wiedza na temat RE może **bardzo** zwiększyć naszą produktywność! Poza tym wyrażenia regularne są uniwersalne. W C#, Javie, Perlu, Pythonie itd. wyglądają niemal tak samo. Możesz je nawet wykorzystać podczas zadań nie związanych z programowaniem (np. do masowej zmiany nazw plików).

Regex vs. metody z klas String oraz Char.

Klasy String i Char zawierają mnóstwo metod, które są przydatne przy przetwarzaniu tekstu. Jednak w niektórych przypadkach ich zastosowanie nie jest zalecane. Przypuśćmy na przykład, że zachodzi konieczność sprawdzenia czy nazwa pliku pasuje do określonego wzorca. Można to zrobić bez użycia wyrażeń regularnych ale takie rozwiązanie ma co najmniej cztery bardzo poważne wady:

- kod jest długi,
- kod jest nieczytelny,
- kod jest **podatny na błędy**,
- kod jest **nieelastyczny**.

Kod jest podatny na błędy ponieważ często polega na odwołaniu do indeksów poszczególnych znaków w nazwie - w takim przypadku nie trudno o pomyłkę, za to ciężko wykryć taki błąd.

Kod jest nieelastyczny ponieważ nawet najmniejsza zmiana formatu nazwy może wiązać się z koniecznością przerobienia **całej** metody sprawdzającej jej poprawność! Gdy do sprawdzania nazwy użyjemy regex wtedy zajdzie konicznośc zmiany **jednej** linijki kodu lub nawet **żadnej**. Wyrażenie sprawdzające nazwę może być przecież ładowane z wiersza poleceń albo pobierane z sieci. Może być nawet tak, że odpowiednie wyrażenie dostarcza klient a my tylko piszemy aplikację, która je wykorzystuje.

Składnia wyrażeń regularnych.

Zapis wyrażeń regularnych jest bardzo bogaty. W tym artykule wymienię tylko jego podstawowe elementy, których opanowanie jest proste i daje duże możliwości. Przedstawione informacje będzie można utrwalić dzięki ćwiczeniom, które umieściłem ma końcu tekstu (podaję składnie już teraz by Czytelnik miał możliwość zrozumienia co dzieje się w omawianych dalej przykładach użycia wyrażeń regularnych w .NET).

Składnia wyrażeń regularnych. Klasy znaków.

Za pomocą klas znaków można określić typ symbolu, który wyrażenie regularne uzna za prawidłowy (pasujący do wzorca).

`.`	Każdy znak z wyjątkiem *\n*. Jeśli ustawiona jest opcja Singleline kropka pasuje też do *\n*.
`[abc]`	Dowolny znak ze wskazanego zbioru.
`[^abc]`	Dowolny znak prócz tych ze zbioru.
`[a-z0-9]`	Dowolny znak z zakresu *a* do *z* lub od *0* do *9*.
`\w`	Dowolny znak z klasy word (litera, cyfra, podkreślnik).
`\W`	Dowolny znak z poza klasy word.
`\s`	Dowolny znak z klasy white-space.

\S	Dowolny znak z poza klasy white-space.
\d	Dowolna cyfra dziesiętna.
\D	Dowolny znak inny niż cyfra dziesiętna

Składnia wyrażeń regularnych. Asercje pozycjonowania.

Przy użyciu poniższych symboli można wyznaczyć miejsce w tekście, w którym musi pojawić się dopasowanie (tekst pasujący do wzorca określonego przez wyrażenie).

^	Dopasowanie musi się pojawić na początku tekstu. Jeśli ustawiona jest opcja Multiline może to być początek linii.
$	Dopasowanie musi się pojawić na końcu tekstu. Jeśli ustawiona jest opcja Multiline może to być koniec linii.*
\b	Dopasowanie musi się znajdować na początku lub końcu słowa.
\B	Dopasowanie nie może się znajdować na początku lub końcu słowa.

* Jest z tym pewien problem (wyjaśnię to w ćwiczeniach).

Składnia wyrażeń regularnych. Powtórzenia.

Za pomocą tych kwantyfikatorów można określić ile razy ma wystąpić dany znak (lub grupa znaków). Symbol powtórzenia odnosi się do elementu, po którym następuje (do pierwszego po jego lewej stronie).

*	Zero lub więcej razy.
+	Jeden lub więcej razy.
?	Zero lub jeden raz.
{n}	Dokładnie *n* razy.
{n,}	Minimum *n* razy.
{n,m}	Minimum *n* maksimum *m* razy.

Składnia wyrażeń regularnych. Greedy vs Lazy.

Przypuśćmy, że stosujemy wyrażenie \bb\w* do znalezienia słów, które zaczynają się na literę *b*. Domyślne zachowanie wyrażeń regularnych jest zachłanne, tzn. wyrażenie dopasuje jak najdłuższe słowo pasujące do wzorca. Możemy to zmienić stosując zamiast kwantyfikatora * jego leniwą wersję: *? Teraz wyrażenie dopasuje jak najkrótsze słowo pasujące do wzorca. W tabelce znajdują się leniwe wersje wcześniej przedstawionych kwantyfikatorów.

*?	Zero lub więcej razy (leniwy).
+?	Jeden lub więcej razy (leniwy).
??	Zero lub jeden raz (leniwy).
{n,}?	Minimum *n* razy (leniwy).
{n,m}?	Minimum *n* maksimum *m* razy (leniwy).

Składnia wyrażeń regularnych. Inne kwestie.

Poniżej przedstawiam kilka innych elementów składni, które z pewnością często się przydają.

\	Ukośnik jest również znakiem ucieczki. Tzn. likwiduje znaczenie znaków specjalnych np. \. znaczy dosłownie „kropka”.
\0x20	Kod szesnastkowy znaku ASCII
\u0020	Kod szesnastkowy znaku Unicode.
#	Wszystko po tym znaku jest komentarzem.*
()	Nawiasy zwykłe używane są do grupowania symboli.
|	Pionowa kreska oznacza alternatywę.

* Zależy od ustawiania opcji (omówię to przy okazji ćwiczeń).

Nie martw się jeśli nie jesteś pewien znaczenia któregoś z przedstawionych elementów syntaktycznych. Wszystko stanie się jasne gdy trochę poćwiczymy!

Regex vs metody z klas String oraz Char. Przykładowy program.

Przewagę wykorzystania wyrażeń regularnych doskonale widać na przykładzie poniższego zadania (realny problem, z którym się zetknąłem).

Trzeba zbudować program będący elementem instalatora, który wykonuje pewne operacje na bazie danych. Aby wykonać te operacje program musi uruchomić skrypty, które znajdują się w określonych plikach. Problem w tym, że aplikacja nie może uruchomić wszystkich plików, które znajdują się w katalogu lecz jedynie te, których nazwa ma określony format. Wywołanie tylko odpowiednich plików ma krytyczne znaczenie - jeśli się pomylimy baza zostanie uszkodzona.

Nazwa plików, które mają zostać wykonane ma taki schemat (case-insensitive):
d_123_xyz_99_data.sql

- dokładnie jedna litera z wyjątkiem a, b oraz c,
- dowolna liczba trzycyfrowa,
- słowo *xyz* lub *pki,*
- dowolna liczba dwucyfrowa,
- dowolne słowo od długości od 2 do 7 znaków (brak spacji).

Jeśli zastosujemy wyrażenia regularne cała metoda sprawdzająca format nazwy ma jedną linijkę kodu:

```
private bool CzyNazwaPoprawnaRegEx(string nazwa)
{
    return Regex.IsMatch(nazwa, wzorzecTextBox.Text, RegexOptions.IgnoreCase);
}
```

Nazwa, którą sprawdzamy jest podawana jako parametr, a jej format jest określany przez wyrażenie pobierane z komponentu TextBox. Znaczenie Regex.IsMatch() oraz RegexOptions.IgnoreCase omówię dokładnie w dalszej części artykułu. Pamiętaj, że podawane przez użytkownika wyrażenie może być nieprawidłowe i wówczas generowany jest wyjątek (jest on jednak obsługiwany w innym punkcie kodu). Oto prawidłowe wyrażenie testujące nazwę: `^[d-z]_\d{3}_(xyz|pki)_\d{2}_\w{2,7}.sql$`

Dla porównania metoda sprawdzająca poprawność nazwy bez zastosowania RE wygląda tak:

```
private bool CzyNazwaPoprawnaSadoMaso(string nazwa)
{
    try
    {
        if (nazwa[0] == 'a' || nazwa[0] == 'b' || nazwa[0] == 'c' ||
            !Char.IsLetter(nazwa[0]))
        {
            return false;
        }

        if (nazwa[1] != '_')
        {
            return false;
        }

        if (!Char.IsDigit(nazwa[2]) || !Char.IsDigit(nazwa[3]) ||
            !Char.IsDigit(nazwa[4]))
        {
            return false;
        }

        if (nazwa[5] != '_')
        {
            return false;
        }

        if (nazwa.Substring(6, 3) != "xyz" && nazwa.Substring(6, 3) != "pki")
```

```
        {
            return false;
        }

        if (nazwa[9] != '_')
        {
            return false;
        }

        if (!Char.IsDigit(nazwa[10]) || !Char.IsDigit(nazwa[11]))
        {
            return false;
        }

        if (nazwa[12] != '_')
        {
            return false;
        }

        if (nazwa.IndexOf('.', 13) - 13 < 2 || nazwa.IndexOf('.', 13) - 13 > 7)
        {
            return false;
        }
        else
        {
            string s = nazwa.Substring(13, nazwa.IndexOf('.', 13) - 13);
            foreach (char i in s)
            {
                if (!Char.IsLetter(i))
                {
                    return false;
                }
            }
        }

        if (nazwa.Substring(nazwa.Length - 4) != ".sql")
        {
            return false;
        }
    }
    catch
    {
        return false;
    }

    // Jeśli wykonanie dojdzie do tego punktu można przyjąć, ze nazwa jest
    // prawidłowa.
    return true;
}
```

No cóż chyba jasno widać, że nie tędy droga?
Pełny kod przykładowego programu (VS 2005) zawierający prosty interfejs i obszerne komentarze jest załączony do artykułu.

Wyrażenia regularne w .NET.

Wyrażenia regularne są częścią biblioteki klas podstawowych. Typy, które je implementują znajdują się w przestrzeni nazw System.Text.RegularExpressions. Ich składnia jest zgodna z tą znaną z Perl 5.

Najważniejszym elementem obsługi RE w .NET jest klasa Regex. Można wykorzystywać jej metody statyczne bądź też utworzyć egzemplarz tej klasy.

Wyrażenie regularne przechowywane jest albo w postaci MSIL (języka pośredniego maszyny wirtualnej), albo w postaci sekwencji wewnętrznych instrukcji (domyślnie), wówczas jest ono interpretowane przez specjalny engine. By RE zostało skompilowane do MSIL a następnie w miarę potrzeby skompilowane (JIT) do kodu natywnego należy użyć opcji RegexOptions.Compiled. Użycie tej opcji zwiększa szybkość wykonywania ale za to zużywa więcej pamięci - nawet wtedy gdy obiekt regex jest już niepotrzebny pamięć jest nadal zajęta (zwalniana jest dopiero przy usuwaniu całego kodu aplikacji). Można

używać prekompilowanych wyrażeń regularnych (zapisanych w DLL), to eliminuje konieczność kompilacji po uruchomieniu programu.

Ponieważ celem artykułu jest zapoznanie Czytelnika z podstawami RE i zachęcenie go do dalszej nauki przedstawię jedynie najpotrzebniejsze elementy trzech najbardziej przydatnych klas: Regex, Match oraz MatchCollection. Po pełniejszy opis tych a także innych klas takich jak Group czy Capture odsyłam do dokumentacji .NET (opisuje te zagadnienia w przystępny sposób).

Zanim zaczniemy warto wspomnieć o problemie zwanym *plagą backslashy*. O co chodzi? W wyrażeniach regularnych znak \ ma specjalne znaczenie. Żeby zlikwidować to specjalne znaczenie należy ten znak podwoić. Niestety w stringach w C# backslash też ma specjalne znaczenie. W celu pozbycia się tego znaczenia należy ten znak... podwoić. W taki sposób by zapisać jeden backslash w wyrażeniu trzeba by użyć w stringu aż czterech! Na szczęście projektanci C# wprowadzili do języka *stringi dosłowne*. Aby zlikwidować specjalną interpretację znaków w łańcuchu wystarczy poprzedzić go znakiem @. Np. `@"\n"` nie oznacza wcale nowej linii tylko napis \n. Oczywiście jeśli chcemy zapisać w wyrażeniu znak \ musimy go podwoić (tego wymaga bowiem składnia wyrażeń regularnych). Napiszemy więc tak: `@"\\"`, na pewno wygląda to lepiej niż zapis: `"\\\\"`.

Wyrażenia regularne w .NET. Klasa Regex.

Z klasy Regex możemy korzystać dzięki jej metodom statycznym bądź też przez utworzenie obiektu tej klasy. Reprezentuje ona niezmienne wyrażenie (po jego utworzeniu nie można go już przekształcić).

Gdy wyrażenia będziemy używać rzadko wygodnie jest skorzystać z metod statycznych. Jeśli natomiast będziemy je wykorzystywać częściej wtedy warto jest utworzyć stosowny obiekt klasy Regex (również ze względu na efektywność).

Przykład:
Użycie metody statycznej do sprawdzenia czy w stringu znajduje się słowo zaczynające się na literę *a*.

```
Console.WriteLine(Regex.IsMatch("dom", @"\ba\w*")); // False
```

Przykład:
Użycie metody obiektu do sprawdzenia czy w stringu znajduje się słowo zaczynające się na literę *a*.

```
Regex re = new Regex(@"\ba\w*");
Console.WriteLine(re.IsMatch("trawa agrest")); // True
```

Wyrażenia regularne w .NET. Klasa Regex. Opcje.

Przy tworzeniu obiektu klasy Regex lub wykorzystywaniu jej metod statycznych można skorzystać z kilku opcji:

`Compile`	Sprawia, że wyrażenie jest kompilowane do MSIL. (szybsze działanie ale dłuższy start i dłużej zajmowana pamięć)
`CultureInvariant`	Określa, że różnice między językami będą ignorowane.
`ECMAScript`	Określa, zachowanie zgodne z ECMAScript. Użycie wraz z nią opcji innych niż Compile, IgnoreCase lub Multiline spowoduje wyjątek.
`ExplicitCapture`	Określa, że nienazwane nawiasy zachowują się jak grupy nieprzechwytujące.
`IgnoreCase`	Sprawia, że wielkość liter nie ma znaczenia.
`IgnorePatternWhitespace`	Eliminuje z wyrażenia białe znaki i w ten sposób umożliwia zwiększenie czytelności oraz stosowanie `# komentarzy`
`Multiline`	Sprawia, że `^` oraz `$` odnoszą się również do początku i końca linii.*
`None`	Oznacza, że nie określono opcji.
`RightToLeft`	Sprawia, że przeszukiwanie obywa się od prawej do lewej strony.
`Singleline`	Określa że . (kropka) pasuje także do znaku nowej linii \n.

* Jest z tym pewien problem (wyjaśnię to w ćwiczeniach).

W celu użycia opcji podajemy je jako parametr przy tworzeniu obiektu lub korzystaniu z metod statycznych.

Do połączenia kilku opcji stosujemy operator | (alternatywa bitowa).

Przykład:
Wykorzystanie opcji IgnoreCase i Multiline:

```
Regex re = new Regex(@"\ba\w*", RegexOptions.IgnoreCase | RegexOptions.Multiline);
```

Jak widać wszystkie opcje należą do wyliczenia RegexOptions.

Wyrażenia regularne w .NET. Regex.Replace().

Jeśli chcemy zastąpić jakimś tekstem wszystkie dopasowania do określonego wyrażenia możemy skorzystać z metody Replace() klasy Regex.

Przykład:
Zastąpienie słów zaczynających się na literę *k* słowem *kotletem*.

```
Console.WriteLine(Regex.Replace("Uderzył go kijem!", @"\bk\w*", "kotletem"));
```

Wynik:
Uderzył go kotletem!

Wyrażenia regularne w .NET. Regex.Split().

Klasa String udostępnia praktyczną metodę podziału tekstu wg jakiegoś znaku. Np. by uzyskać czteroelementową tablicę typu string zawierającą poszczególne składowe adresu IP można wykorzystać taki kod:

```
string[] ip = "127.0.0.1".Split('.');
```

Podobny efekt można otrzymać korzystając z metody Split() klasy Regex. W przypadku użycia tej metody tekst zostanie podzielony zgodnie z wystąpieniem dopasowania.

Przykład:
Stworzenie tablicy, której pierwszy element będzie zawierał nazwę zmiennej a drugi jej wartość (zakładamy, że są one rozdzielone pascalowym operatorem przypisania wraz ze spacjami):

```
string[] s = Regex.Split("X := 1234", " := ");
```

Jeśli zastosujemy wyrażenie regularne, które może dopasować pusty tekst, wówczas w wyniku użycia Regex.Split() otrzymamy tablicę pojedynczych znaków. Stanie się tak dlatego, że pusty string możemy odnaleźć w każdym punkcie badanego łańcucha. Dodatkowo oprócz znaków, które wystąpiły w badanym tekście, na początku i na końcu tablicy wystąpi pusty string.

Przykład:
Kod tworzący pięcioelementową tablicę:

```
string[] s = Regex.Split("123", "a*");
```

Wartości w tej tablicy będą wyglądać tak:
""
"1"
"2"
"3"
""

Wyrażenia regularne w .NET. Klasa Match.

Klasa Match reprezentuje wynik dopasowania (dopasowanie to fragment badanego tekstu, który pasuje do określonego wzorca). W celu utworzenia obiektu tej klasy należy skorzystać z metody Match() klasy Regex.

Przykład:
Zbadanie istnienia słowa zaczynającego się na literę *a*:

```
Regex re = new Regex(@"\ba\w*");
Match m = re.Match("mysz agrest")
```

Za pomocą właściwości Success możemy sprawdzić czy nastąpiło dopasowanie:

```
Console.WriteLine(m.Success); // True
```

Właściwość Value pozwala na uzyskanie tekstu dopasowania:

```
Console.WriteLine(m.Value); // agrest
```

Taki sam wynik da użycie m.ToString().

Przy użyciu Index możemy pobrać miejsce, w którym zaczyna się dopasowanie (liczone od 0):

```
Console.WriteLine(m.Index); // 5
```

Użycie Length podaje długość dopasowania:

```
Console.WriteLine(m.Length); // 6
```

Jeśli w tekście jest więcej niż jedno dopasowanie obiekt zwrócony przez RegEx.Match() odnosi się do pierwszego z nich.

Wyrażenia regularne w .NET. Klasa MatchCollection.

Klasa MatchCollection reprezentuje sekwencje dopasowań zwróconą przez metodę Matches() klasy Regex.

Przykład:
Wykorzystanie klasy MatchCollection do znalezienia wszystkich słów zaczynających się na literę *a*:

```
Regex re = new Regex(@"\ba\w*");
MatchCollection mc = re.Matches("mysz agrest ameba garnek tama analogia");
Console.WriteLine("Znaleziono: {0}", mc.Count);
foreach (Match i in mc)
{
    Console.WriteLine("{0} {1} {2}", i, i.Length, i.Index);
}
```

Wynik:
Znaleziono: 3
agrest 6 5
ameba 5 12
analogia 8 30

Czy zawsze warto używać wyrażeń regularnych?

Należy sobie uświadomić, że wyrażenia regularne nie nadają się do wszystkich zadań. Świetnie sprawdzają się w problemach, w których istotna jest jedynie syntaktyka (składnia). Jeśli natomiast oprócz tego jak wygląda tekst ważne jest także co ten tekst oznacza (jego semantyka) warto użyć innych technik niż RE.

Dobrze widać to na przykładzie wyrażenia, które sprawdza poprawność adresu IPv4:
```
^(2[0-4]\d|25[0-5]|[01]?\d\d?)\.(2[0-4]\d|25[0-5]|[01]?\d\d?)\.(2[0-4]\d|25[0-5]|[01]?\d\d?)\.(2[0-4]\d|25[0-5]|[01]?\d\d?)$
```
Prawda, że to trochę nieporęczne? Jego skomplikowanie związane jest z tym iż **wartość** każdej z czterech liczb musi być w przedziale 0..255.

Regex + String + Int. Przykładowy program.

Czasem warto pomyśleć o połączeniu wykorzystania wyrażeń regularnych wraz z metodami z takich klas jak String, Int, Char...

O zaletach takiego rozwiązania można się przekonać na podstawie przykładowego programu, który sprawdza poprawność kropkowo-dziesiętnego zapisu adresu IPv4 (np. ten adres jest dobry: 127.0.0.1, ale ten już nie: 122.256.0.0).

Poniższa metoda używa Regex.IsMatch() do sprawdzenia czy podany tekst składa się z 4 liczb odzielonych kropkami. Jeśli tak jest to przy użyciu adres.Split('.') uzyskujemy dostęp do poszczególnych składowych adresu. Następnie sprawdzamy czy wartości żadnej składowej nie przekracza 255. Nie sprawdzamy dolnej granicy ponieważ użycie wyrażenia regularnego upewnia nas, iż między kropkami znajdują się cyfry (więc może tam być min. 0).

```
private bool CzyAdresIPv4JestPoprawny(string adres)
{
    bool adresOK = true;

    if (Regex.IsMatch(adres, @"^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$"))
    {
        string[] iP = adres.Split('.');
        if (int.Parse(iP[0]) > 255 || int.Parse(iP[1]) > 255 ||
            int.Parse(iP[2]) > 255 || int.Parse(iP[3]) > 255)
        {
            adresOK = false;
        }
    }
    else
    {
        adresOK = false;
    }

    return adresOK;
}
```

Pełny kod programu (VS 2005) jest załączony do artykułu.

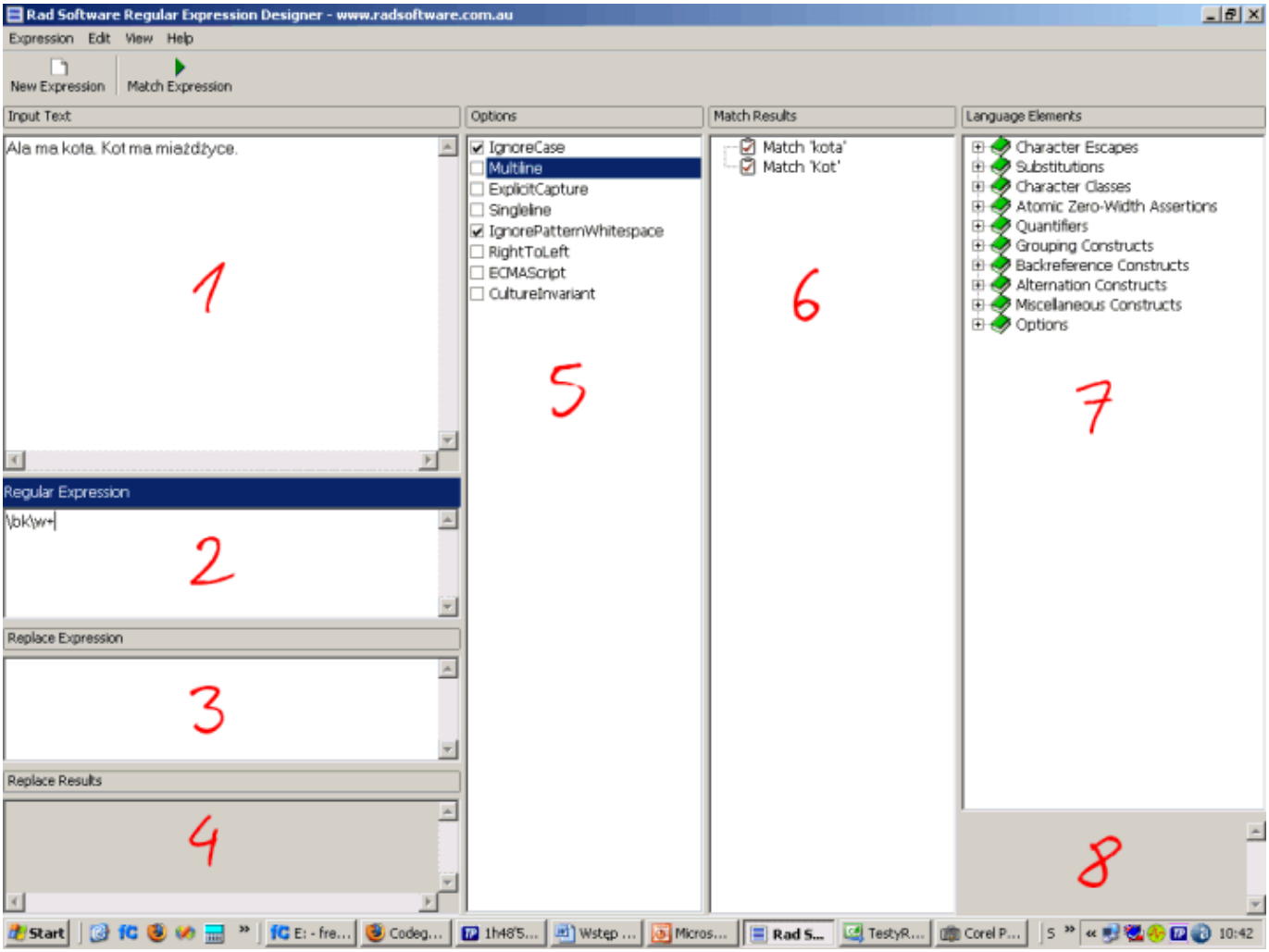
Oczywiście skoro ktoś zadał sobie trud stworzenia i opublikowania wyrażenia sprawdzającego adres IP nic nie stoi na przeszkodzie by je po prostu skopiować. W tym przykładzie jednak chodziło mi o ukazanie tego, że RE **nie nadają** się zbytnio do sprawdzania jaką **wartość** ma fragment tekstu.

Ćwiczenia.

Nie można nauczyć się wyrażeń regularnych przez samo patrzenie na ich pogmatwaną składnie. Należy nieco potrenować. Do ćwiczeń wykorzystamy darmowy program Rad Software Regular Expression Designer 1.2 (wymaga NET 1.1). Można też skorzystać z darmowego The Regulatora lub shareware'owego Expresso.

Program można pobrać z tej strony: http://www.radsoftware.com.au/regexdesigner

Regular Expression Designer jest bardzo prosty w obsłudze, nie zaszkodzi jednak mały opis tego do czego służą poszczególne okna.



1 - W to okno wpisujemy tekst, na którym będziemy testować wyrażenie regularne.
2 - Tutaj wpisujemy wyrażenie.
3 - W tym oknie wpisujemy (opcjonalnie) tekst, który zostanie wstawiony do wejściowego tekstu w miejscu dopasowania.
4 - Tutaj możemy zobaczyć rezultat zastąpienia.
5 - W tym oknie możemy ustawić opcje wyrażenia regularnego.
6 - Tu możemy zobaczyć rezultaty dopasowania.
7 - W tym oknie widzimy elementy składniowe, które możemy użyć w wyrażeniu.
8 - Gdy zaznaczymy jakiś element składniowy, w tym miejscu pojawi jego się jago szerszy opis.

Aby uruchomić sprawdzanie tekstu naciskamy przycisk z zieloną strzałką (lub klawisz F5).

Uwaga!
Podczas kopiowania zamieszczonych tu przykładów wyrażeń regularnych uważaj by wraz z nimi nie skopiować niepotrzebnych białych znaków...

Ćwiczenie 1.
Tekst:
Duży dom.
Stefan jest domniemanym sprawcą tego czynu.
DOM (Document Object Model)

1.1
Wyrażenie: `dom`

Chociaż może to się wydawać dziwne, słowo `dom` jest w 100% poprawnym regex! Dopasowuje ono dokładnie ciąg symboli *dom* (lub jakąś jego odmianę jeśli ustawimy ignorowanie wielkości liter). Tak więc w zależności od opcji IgnoreCase powinieneś otrzymać 2 lub 3 dopasowania (zaznacz dowolne z nich żeby zobaczyć, w którym miejscu tekstu ono wystąpiło).

1.2
Zauważ, że `dom` dopasowuje również tekst będący częścią słowa *domniemanym*. Aby wyrażenie akceptowało jedynie *dom* będący osobnym słowem należy zmienić nieco wyrażenie przez użycie asercji `\b`, która oznacza koniec lub początek słowa.

Wyrażenie: `\bdom\b`

Teraz ciąg znaków w słowie *domniemanym* się nie załapał.

Ćwiczenie 2.
Tekst:
a\aplikacja
a10 a_x
Afganistan
bilon
kompilator

2.1
Wyrażenie: `a.`

Takie wyrażenie dopasuje każdy ciąg symboli składający się z litery *a*, po której następuje dowolny symbol. Kropka oznacza cokolwiek z wyjątkiem znaku nowej linii lub wszystko bez wyjątku (jeśli ustawiona jest opcja Singleline).

2.2
Wyrażenie: `a.*`
Tym razem na końcu wyrażenia jest gwiazdka, która mówi, że dowolny znak (wyznaczony przez kropkę) może powtórzyć się zero lub więcej razy.

2.3
Wyrażenie: `a\d+`

Teraz jedyne dopasowanie to *a10*. Dlaczego? Ponieważ po literze *a* ma wystąpić co najmniej jedna cyfra.

2.4
Wyrażenie: `a\d*`

Wystarczy, że zmienimy kwantyfikator `+` (jeden lub więcej razy) na `*` (zero lub więcej razy) a wynik dopasowania jest diametralnie inny. Tym razem załapią się wszystkie pojedyncze litery *a* oraz ciąg *a10*.

Ćwiczenie 3.
Tekst:
00_233 00-999 34-700 34-99d 433-234

3.1
Wyrażenie: `\d{2}-\d{3}`

To wyrażenie sprawdza poprawność kodu pocztowego. Szuka dwóch cyfr, potem średnika a na koniec znowu trzech cyfr. Czy jednak na pewno to regex działa jak należy?

3.2
Wyrażenie: `\b\d{2}-\d{3}\b`

Poprzednia reguła miała istoty błąd. Akceptowała też cześć tekstu *4**33-234***. Dzięki użyciu `\b` na początku i na końcu wyrażenia likwidujemy ten problem.

Ćwiczenie 4.
Tekst:
14.12.1924 Jan Kowalski
02.01.1999 Stefan Śmietana

24.10.1938 Paweł Nowak
04.09.2002 Anna Kotlet
30.08.2006 Henryk Ziemniak

4.1
Wyrażenie: `.{6}19.*`

Za pomocą tej reguły możemy znaleźć wszystkie osoby urodzone w latach 1900..1999. Jak to działa? Najpierw może wystąpić sześć dowolnych znaków a potem ma wystąpić ciąg *19* po którym może być dowolny tekst. Zauważ, że (w okienku z dopasowaniami) na końcu każdego nazwiska widać kwadracik. Pojawia się on dlatego, że kropka pasuje również do \r (carriage return).

4.2
Wyrażenie: `.{6}19[\w ]*`

Tym razem brzydki kwadracik znikł. Dlaczego? Ponieważ teraz stwierdzamy, że po ciągu *19* ma wystąpić znak należący do klasy word (litera, cyfra lub podkreślnik) albo spacja. Zauważ, że spacja podana w nawiasach prostokątnych jest rozumiana jako spacja niezależnie od ustawienia opcji IgnorePatternWhitespace.

Ćwiczenie 5.
Tekst:
ananas banan gra proces atv bbc exe

5.1
Wyrażenie: `[ab]..`

Ta reguła określa ciąg trzech symboli, z których pierwszy jest albo literą *a* albo literą *b*, kolejne dwa symbole są dowolne.

5.2
Wyrażenie: `[^ab]..`

Tym razem szukamy ciągu trzech symboli, takiego, że pierwszy symbol nie jest ani literą *a* ani literą *b*.

Ćwiczenie 6.
Tekst:
ananas banan gra proces atv bbc execement dom
traktor 24h 74lata

6.1
Wyrażenie: `\b[a-d]\w*`

Dzięki tej regule znajdziemy wszystkie słowa zaczynające się na literę *a*, *b*, *c* lub *d*. Zauważ, że średnik gdy jest wewnątrz nawiasów prostokątnych zachowuje się jak operator zakresu. Zwróć także uwagę, że na końcu wyrażenia nie umieściłem `\b`. Nie trzeba tego robić ponieważ `\w` nie zaakceptuje spacji (która przecież rozdziela słowa).

6.2
Wyrażenie: `\b[a-d0-5]\w*`

Tym razem szukamy słów, które zaczynają się na *a*, *b*, *c*, *d*, *0*, *1*, *2*, *3*, *4*, lub *5*.

Ćwiczenie 7.
Tekst:
anakonda ma nos jak antena

7.1
Wyrażenie: `ana?`

Otrzymaliśmy dwa dopasowania (*ana*, *an*). Stało się tak ponieważ `?` oznacza, że poprzedzająca go litera *a* może pojawić się dokładnie raz lub ani razu.

7.2
Wyrażenie: `a(na)?`

Umieszczając ciąg *na* w nawiasach określamy, że cały ciąg *na* może wystąpić zero lub jeden raz. Podobnie możemy użyć nawiasów z każdym innym kwantyfikatorem, np. z `*` Jeśli nie masz włączonej opcji ExplicitCapture w oknie z dopasowaniami pojawiły się elementy z klas Group i Capture (ich omówienie wykracza poza ramy tego artykułu).

Ćwiczenie 8.
Tekst:
sysdm.cpl
examplefile.html
boot.ini
services.msc

8.1
Wyrażenie: `\b\w{1,8}\.\w{3}\b`

Za pomocą tej reguły sprawdzamy czy nazwa ma format 8.3. Najpierw oznaczamy, że ma wystąpić od 1 do 8 znaków z klasy word (z których pierwszy znajduje się na granicy słowa). Potem ma wystąpić kropka (dosłownie kropka więc poprzedziliśmy ją znakiem ucieczki `\`). Na koniec mają wystąpić dokładnie 3 znaki z klasy word.

8.2
Wyrażenie:
```
\b # granica slowa
\w{1,8} # od 1 do 8 znakow
\. # kropka
\w{3} # 3 znaki
\b # granica slowa
```

Aby móc skorzystać z takiej formy wyrażenia regularnego musisz ustawić opcję IgnorePatternWhitespace. Jej ogromną zaletą jest to, że możemy tworzyć komentarze przy użyciu znaku `#`.

Ćwiczenie 9.
Tekst:
kot dom kotwica domowa
GetPrice

9.1
Wyrażenie: `kot|dom`

Ta reguła dopasowuje wszystkie wystąpienia ciągu *kot* lub *dom*.

9.2
Wyrażenie: `\b(kot|dom)\b`

Za pomocą tego wyrażenie znajdziemy wszystkie słowa *kot* lub *dom*. W odróżnieniu od punktu 9.1 cześć *kot* w słowie *kotwica* nie zostanie dopasowana. Koniecznie sprawdź jak zmieni się działanie wyrażenia po usunięciu nawiasów.

9.3
Wyrażenie: `Get|GetPrice`

Tym razem jedyne dopasowanie to *Get*.

9.4
Wyrażenie: `GetPrice|Get`

Ta reguła zwraca również jedno dopasowanie, ale tym razem jest to ciąg *GetPrice*. Wniosek? Kolejność elementów w alternatywie ma znaczenie! Wyrażenie regularne przestaje szukać kolejnego dopasowania gdy znajdzie już dopasowanie do jakiegoś członu alternatywy.

Ćwiczenie 10.
Tekst:
babaa barłka banan

10.1
Wyrażenie: `b\w+`

Zastosowaliśmy zachłanną wersję kwantyfikatora `+` więc wyrażenie szuka najdłuższego możliwego dopasowania.

10.2
Wyrażenie: `b\w+?`

Tym razem zmieniliśmy kwantyfikator na leniwy więc wyrażenie dopasuje jak najkrótszy tekst, który pasuje do wzorca (w tym przypadku będzie to litera *b* i jeden znak z klasy word).

Ćwiczenie 11.
Tekst:
to chyba
jest jakaś
szansa

11.1
Wyrażenie: `\b\w*\w$`
Jeśli mamy wyłączoną opcje Multiline powinno zostać dopasowane jedynie słowo: *szansa*. Jest tak dlatego, że symbol `$` oznacza w tym przypadku koniec tekstu. Co się zmieni gdy ustawimy opcje Multiline? Niestety nic! Jeśli masz zainstalowanego Pythona spróbuj jak zadziała to z programem redemo.py (jest dołączany przy standardowej instalacji w katalogu Tools\Scripts). Ustaw opcje Multiline oraz zaznacz opcje Highlight all matches i przekonaj się, że zostały znalezione trzy dopasowania: *chyba*, *jakaś* oraz *szansa*.

11.2
Wyrażenie: `\b\w*[\w\r]$`
Ustawiamy opcje Mutiline - tym razem udało się nam dopasować słowa *chyba*, *jakaś* oraz *szansa*. Stało się tak dlatego, że teraz na końcu linii może znajdować się znak z klasy word albo znak *\r* (carriage return).

Podsumowanie.

Wyrażenia regularne są potężnym narzędziem, które w wielu przypadkach pozwala na tworzenie elastycznego, klarownego i efektywnego kodu. Nie każdy programista .NET musi być znawcą tego złożonego tematu, ale każdy powinien chociaż rozumieć podstawy ich składni i sposobu wykorzystania na platformie .NET Framework. Jeśli chcesz zaoszczędzić sobie w przyszłości wiele czasu i nerwów zainwestuj jeden lub dwa dni na opanowanie tych podstaw - z pewnością nie pożałujesz!

Jeśli masz trochę więcej czasu i chęci poszerz swoją wiedzę na temat RE - zwróć szczególną uwagę na takie elementy jak konstrukcje grupujące, wzory zastępowania czy odwołania wstecz...

Kod dołączony do artykułu można pobrać z tego adresu:
http://morzel.net/edu/wdre/wdre_src.zip