DELTA – Střední škola informatiky a ekonomie, s.r.o.
Ke Kamenci 151, PARDUBICE

# MATURITNÍ PROJEKT

# *KURZ REGULÁRNÍCH VÝRAZŮ*

Příjmení, jméno: Unčovský, Josef
Třída: 4.B
Studijní obor: Správce informačních systémů
Školní rok: 2020/2021

# Zadání maturitního projektu z informatických předmětů

| | |
|---|---|
| Jméno a příjmení: | *Josef Unčovský* |
| Školní rok: | *2020/2021* |
| Třída: | *3.B* |
| Obor: | *Informační technologie 18-20-M/01* |
| | |
| Téma práce: | *Tutoriál a sbírka cvičení pro výuku regulárních výrazů* |
| Vedoucí práce: | *RNDr. Jan Koupil, Ph.D.* |

## Způsob zpracování, cíle práce, pokyny k obsahu a rozsahu práce:

Cílem práce je vytvořit stručný studijní text popisující regulární výrazy doprovázený bohatou nabídkou jednak ukázkových, jednak neřešených cvičení, na kterých si žák ověří své porozumění tématu i řemeslnou zručnost. Těžiště projektu bude ležet v těchto cvičeních, nikoli textu samotném, přičemž cvičení by měla být do vysoké míry autonomní – tedy sama se opravující, resp. ohlašující chyby.

1. Žák podnikne rešerši za účelem nalezení existujících návodů a tutoriálů k výuce regulárních výrazů a také vhodných nástrojů k jejich zkoušení (online prostředí i offline s použitím unit-testů). Zaměří se na jeden ze tří jazyků, které se na škole Delta vyučují, tedy C#, JavaScript nebo PHP.
2. Žák sestaví kurz regulárních výrazů tak, aby postupně přidávané symboly a jejich použití pokračovaly od nejjednodušších ke složitějším a jejich kombinování. Ke každému příkazu, či malé sadě příkazů připraví několik ukázkových použití a také cvičení. Cvičení a případně i ukázky budou využívat nástrojů nalezených v 1. úkolu tak, aby cvičení žákům dávala zpětnou vazbu, zda se podařilo úkol vyřešit, či nikoliv a v maximální míře též informaci, kde řešení selhává.
3. Kromě jednotlivých cvičení cílených na jediný jev či symbol budou vytvořeny také komplexnější úlohy, kterými budou jednoduchá cvičení proložena tak, aby si žák ověřil, že již probrané části ovládá a je schopen je použít samostatně bez kontextu, o který konkrétní symbol se v řešení jedná.
4. V závislosti na čase a tématech probíraných v nižších ročnících otestuje žák, pokud to bude možné, alespoň části svého tutoriálu a formou osobní zpětné vazby nebo dotazníku zjistí, zda je zvolený přístup vhodný, a případně jej upraví.

Dokumentace práce bude odrážet tyto úkoly. V teoretické části bude popsáno, co regulární výrazy jsou a jak se provádí jejich vyhodnocování, dále žák popíše vlastnosti zvoleného prostředí k výuce/testování, poté budou uvedeny vlastní studijní texty včetně několika ukázek zadání cvičení a v poslední kapitole budou shrnuty zkušenosti s použitím, pakliže se tutoriál podaří alespoň částečně ověřit.

## Stručný časový harmonogram (s daty a konkretizovanými úkoly):

- *Září*: Rešerše existujících řešení a vhodných nástrojů
- *Říjen*: Ladění testovacího prostředí, první jednoduché ukázky
- *Listopad*: Návrh struktury tutoriálu, členění kapitol, plnění prvních části
- *Prosinec*: Plnění tutoriálu obsahem
- *Leden*: Testování s uživateli (bude-li situace ve škole připouštět)
- *Únor*: Úpravy UI a navržených cvičení na základě výsledků předchozích testů, opakované testy, práce na dokumentaci projektu
- *Březen*: Práce na dokumentaci projektu

Prohlašuji, že jsem maturitní projekt vypracoval(a) samostatně, výhradně s použitím uvedené literatury.

V Pardubicích 31.3.2020

*(vlastnoruční podpis)*

## Anotace:

Cílem projektu je vytvořit kurz, který prohloubí znalosti uživatele v oblasti regulárních výrazů.

## Klíčová slova:

Kurz; JavaScript; Regulární výrazy; Jasmine; Unit-testy; Repl.it

## Annotation:

The main goal of this project is to create a course, which will increase the knowledge of the user in the topic of regular expressions.

## Keywords:

Course; JavaScript; Regular expressions; Jasmine; Unit-tests; Repl.it

## Poděkování

Děkuji panu RNDr. Janu Koupilovi, Ph.D. za odborný pohled z hlediska vývoje projektu.

# Obsah

# 1: Úvod

Většina dnešních programovacích jazyků (kupříkladu Python, Perl, C#, Javascript, PHP, Java, Ruby a další) nabízí knihovny regulárních výrazů, nebo mají dokonce jejich podporu zabudovanou přímo do jazyka. Každý jazyk přidává vlastní „příchuť" – syntaxi a další vlastnosti. Programátor může ve svých aplikacích pomocí regulárních výrazů nabídnout koncovým uživatelům možnost filtrovat a hledat v datech aplikace. Dalo by se říct, že programátor, který regulární výrazy ovládá, disponuje velice mocným nástrojem. Vyplatí se je tedy naučit. [1]

Všechny UNIXové operační systémy také disponují nástrojem zvaným „grep", který se používá v příkazovém řádku. Tento nástroj prohledává bloky textu a hledá shodu s regulárním výrazem. [2]

Cílem celého projektu je vytvořit celistvý kurz zaměřený na regulární výrazy a jejich implementaci ve skriptovacím jazyce JavaScript. Mou snahou bylo, aby mnou vytvořené příklady co nejlépe ilustrovaly použití regulárních výrazů v praxi (například detekce validní emailové adresy nebo nahrazovaní většího výskytu daného řetězce v textu). Regulární výrazy, které jsou správně použité v praxi, můžou ušetřit spoustu času a zautomatizovat hodně jinak časově náročných úkonů.

# 2: Co jsou regulární výrazy

## 2.1: Definice

V kontextu tohoto projektu je regulární výraz specifický vzor textu, který může programátor použít s mnoha moderními aplikacemi a programovacími jazyky. Programátor může pomocí regulárního výrazu určit, jestli vstup odpovídá vzoru textu. Může také určit, jestli se vzor textu vyskytuje (a kolikrát) v obsáhlejším textu. Také může specifický vzor textu nahradit jiným textem. [1]

Využívají je webové vyhledávače stejně jako textové editory (typicky dialogové okno najít a nahradit). Ve valné většině webových vyhledávačů a textových editorech je služba používající regulární výrazy pod klávesovou zkratkou CTRL+F (v případě textových editorů a vývojových prostředí také CTRL+H pro nahrazení).

Zmíněná vývojová prostředí a některé textové editory nabízejí také možnost po stisknutí CTRL+F nebo CTRL+H vepisovat také samotné regulární výrazy, pokud hledáme více řetězců najednou (Např. pokud potřebujeme najít všechna čísla v souboru, se kterým právě pracujeme).

Regulární výrazy vymyslel v padesátých letech americký matematik Stephen Cole Kleene.[3]

## 2.2: Regulární výrazy v jazyce JavaScript

Jak již bylo zmíněno, každý programovací jazyk přidává regulárním výrazům vlastní „příchuť“. V JavaScriptu definujeme regulární výraz mezi dvěma lomítky, takto:

```
let pattern = /abc/;
```

Obrázek 1 - Definování reg. výrazu v JS

Nebo je možné vytvořit instanci objektu RegExp:

```
let pattern = new RegExp('abc');
```

Obrázek 2 - Definování pomocí instance

## 2.3: Metody

V JavaScriptu má programátor k dispozici několik metod, které může pro práci s regulárními výrazy využít. Konkrétně to je metoda test() objektu RegExp a metody match(), replace(), replaceAll(), search() a split() objektu String. [4]

### 2.3.1: Použití metod (1. Část)

První z metod je metoda matchAll(). Ta vrací iterátor všech výsledků shody řetězce s regulárním výrazem, včetně skupin zachycení (v poli), důležité je za reg. výraz přidat regex flag „g“, jinak metoda vrací chybu.[5] Na obrázku 3 vidíme příklad použití metody v kódu.

```
const regexp = /abc/g;
const string = "abcabc";


const array = [...string.matchAll(regexp)];
console.log(array[0]);
//Output:["abc", index: 0, input: "abcabc", groups: undefined]
```

Obrázek 3 - Metoda matchAll()

### 2.3.2: Použití metod (2. Část)

Druhou metodou je metoda test(). Tato metoda vyhledá shodu mezi řetězcem a regulárním výrazem. Vrací logické hodnoty „true“ nebo „false“.[6] Tato metoda je použita v kurzu regulárních výrazů, přesněji ve zkouškové části. Obrázek 4 ukazuje použití metody v kódu.

```
const regexp = /abc/;
const frst_string = "abc";
const scnd_string = "regex";
console.log(regexp.test(frst_string));
//Output: true
console.log(regexp.test(scnd_string));
//Output: false
```

Obrázek 4 - Metoda test()

### 2.3.3: Použití metod (3. Část)

Třetí metodou je metoda match(). Tato metoda je ve své podstatě omezenou verzí metody matchAll() s tím rozdílem, že nevrací skupiny zachycení (capture groups). Regex flag „g“ zde není nutný. [6] Na obrázku 5 je použití metody v kódu.

```
const string = "Method testing sentence number 1.";
const regex = /[0-9]/g;
const found = string.match(regex);
console.log(found);
//Output: Array ["1"]
```

Obrázek 5 - Metoda match()

## 2.4: Syntaxe

Syntaxi regulárních výrazů se není tak těžké naučit. Skutečnou výzvou je naučit se, jak tuto syntaxi aplikovat a jak rozebrat daný problém, tak, aby se dal vyřešit regulárním výrazem. Další výzvou je fakt, že samotné regulární výrazy (pokud jsou delší) jsou velice nepřehledné. Toto nejlépe ilustruje obrázek 6, na kterém je regulární výraz pro zachycení znaků z množiny „$()*+.?[\^{|“.

```
const regexp = /[\$\(\)\*\+\.\?\[\\\^\{\|]{1,}/;
```

Obrázek 6 - Příklad syntaxe

I někomu, kdo zná syntaxi regulárních výrazů, bude nejspíš chvíli trvat, než se ve výrazu zorientuje a bude si jistý tím, co přesně tento konkrétní regulární výraz zachytává.

Na obrázku 7 je krok za krokem vysvětleno, co zachytává regulární výraz na obrázku 6.

```
[       //Začátek skupiny zachycení
\$      //Zachytí znak "$"
\(      //Zachytí znak "("
\)      //Zachytí znak ")"
\*      //Zachytí znak "*"
\+      //Zachytí znak "+"
\.      //Zachytí znak "."
\?      //Zachytí znak "?"
\[      //Zachytí znak "["
\\      //Zachytí znak "\"
\^      //Zachytí znak "^"
\{      //Zachytí znak "{"
\|      //Zachytí znak "|"
]       //Konec skupiny zachycení
{1,}    //Pokud alespoň jeden znak ze skupiny není obsažen v řetězci
, nebude zachycen
```

Obrázek 7 - Vysvětlení syntaxe

### 2.4.1: Co všechno se dá zachytit?

Jak již bylo zmíněno, regulární výrazy jsou velmi mocný nástroj. Z velké části proto, co všechno mohou v textu zachytit. Odpověď totiž zní „cokoliv“. Regulární výrazy mohou zachytit (nebo nahradit, záleží na situaci) sebekomplexnější text. Datum, IP adresy, telefonní čísla (a varianty všech zemí), emailové adresy, URL, HTML tagy a komentáře, čísla kreditních karet a také konce řádku a tabulátory.

### 2.4.2: Příznaky regulárních výrazů

V podkapitole „Metody“ byly zmíněny „regex flagy“. (Někdy se používá český termín „příznaky“, v tomto textu ale bude použito anglické slovo flag pro regulární výrazy běžně využívané i v češtině.) Jejich úkolem je upravovat režim vyhledávání samotného regulárního výrazu. V JS je jich 6. Konkrétně jsou to flagy „i“ „g“ „m“ „s“ „u“ a „y“.[8] Použití regex flagu je ilustrováno na obrázcích 3 a 5, píše se za regulární výraz (za 2 lomítka).

#### 2.4.2.1: Regex flag „i“

Prvním regex flagem, kterým JS disponuje, je regex flag „i“. Tento flag upraví regulární výraz tak, že změní citlivost na velká a malá písmena. V praxi to znamená, že výraz pak nezná rozdíl mezi „A“ a „a“.

#### 2.4.2.2: Regex flag „g“

Druhým regex flagem, kterým JS disponuje, je regex flag „g“. Tento flag upraví regulární výraz tak, že výraz zachytí všechny shody v řetězci. Regulární výrazy standardně zachytí pouze první shodu v řetězci. Tento regex flag je jeden z nejvíce používaných a některé RegExp metody jeho použití vyžadují.

Na obrázku 8 je vidět zachycení shody regulárního výrazu v textu bez flagu „g“. Regulární výraz tak zachytí pouze první shodu.

```
'regex regex regex'.match(/regex/);
//Output: ["regex", index: 0, input: "regex regex regex", groups: un
defined]
```

Obrázek 8 - nepoužití flagu "g"

Na obrázku 9 je identický regulární výraz, avšak s použitím flagu „g“. Tento výraz zachytí všechny shody v řetězci.

```
'regex regex regex'.match(/regex/g);
//Output: ["regex", "regex", "regex"]
```

Obrázek 9 - použití flagu "g"

### 2.4.2.3: Regex flag „m“

Třetím regex flagem, kterým JS disponuje, je regex flag „m“. Tento flag upraví regulární výraz tak, že znaky „^“ a „$“ (znaky, které v syntaxi regulárních výrazů znamenají začátek a konec řetězce) nezachytávají jen začátek a konec řetězce, ale také začátek a konec řádku. Na obrázcích 10 a 11 je ilustrován rozdíl mezi použitím a nepoužitím tohoto flagu v praxi.

```
string = `1. místo
2. místo
3. místo`
const regexp = /^\d/g;
//Output: 1
```

Obrázek 10 - Nepoužití flagu "m"

Výraz na obrázku 10 nezachytil všechna čísla, protože flag „g“ nefunguje na nové řádky v řetězci.

```
string = `1. místo
2. místo
3. místo`
const regexp = /^\d/gm;
//Output: 1, 2, 3
```

Obrázek 11 - Použití flagu "m"

Výraz na obrázku 11 zachytil všechna čísla. Pro obdobné situace (Je potřeba zachytit všechna čísla na všech řádcích a na každém řádku alespoň jedno je) je nutné použít jak flag „g“, tak flag „m“. Použitím obou těchto flagů regulární výraz pokryje skutečně celý řetězec a najde v něm všechny shody.

### 2.4.2.4: Regex flag „s“

Regex flag „s“ mění chování regulárního výrazu. Přesněji rozšiřuje funkci znaku „.“ (tečka). Tečka v regulárním výrazu, který nepoužívá flag „s“ znamená „jakýkoliv znak“. Symbol tečka standardně nezachytí znak pro nový řádek (newline) – „\n“. Regex flag „s“ rozšíří funkci tečky tak, že zachytí i znak pro nový řádek. Na obrázku 12 a 13 je vidět rozdíl ve funkci regulárních výrazu při použití a nepoužití flagu „s“.

```
'Reg\nex'.match(/Reg.ex/);
//Output: null
```

Obrázek 12 - Nepoužití flagu "s"

Na obrázku 12 vidíme, že regulární výraz selhal. Stalo se tak proto, že tečka ve výchozím režimu nezachytí znak pro nový řádek. (Jak již bylo zmíněno).

```
'Reg\nex'.match(/Reg.ex/s);
//Output["Reg↵ex", index: 0, input: "Reg↵ex", groups: undefined]
```

Obrázek 13 - Použití flagu "s"

Na obrázku 13 vidíme stejný regulární výraz. S použitím flagu „s“ výraz zachytí hledaný řetězec.

### 2.4.2.5: Regex flag „u“

Regex flag „u“ upravuje daný regulární výraz tak, že podporuje celý Unicode. Je ale zvláštní v tom, že funguje pouze se třídou \p (Ta se píše výhradně jako \p{...}, a znamená „properties“, tedy vlastnosti určitého znaku, například písmeno patří do nějaké abecedy a číslo může být arabské nebo čínské. Tato třída tyto informace obsahuje), použití regex flagu „u“ tedy není moc obvyklé.

### 2.4.2.6: Regex flag „y“

Regex flag „y“ upravuje daný regulární výraz tak, že vyhledává shodu na určité pozici. Používá se jen s property „lastIndex“. Ta umožňuje regulárnímu výrazu začít hledat shodu od určité pozice v textu. Na obrázcích 14 a 15 je ilustrováno použití flagu „y“ v praxi.

```
let regexp = /\w+/y;
let str = "let hodnota = '4'";
regexp.lastIndex = 3;
alert(regexp.exec(str));
//Output: null
//(Na pozici 3 je mezera, ne písmeno)
```

Obrázek 14 – použití flagu „y“ na prázdné pozici

```
let regexp = /\w+/y;
let str = "let hodnota = '4'";
regexp.lastIndex = 4;
alert(regexp.exec(str));
//Output: hodnota
//(Na pozici 4 začíná slovo hodnota)
```

Obrázek 15 – použití flagu „y“ na začátku slov

## 3: Struktura kurzu reg. Výrazů


Obrázek 16 – hlavní stránka kurzu reg. výrazů

Na obrázku 16 je snímek hlavní stránky mého kurzu regulárních výrazů. Celkem obsahuje 37 různých příkladů a (jak již bylo zmíněno v úvodu) tyto příklady by měly ilustrovat skutečné použití regulárních výrazů. Tak, jak se s nimi vývojář setká v praxi. Kurz je aktivní na adrese https://regexkurz.000webhostapp.com/ a na hlavní stránce je index příkladů. Odkazy na této stránce vedou na stránku nevyřešeného příkladu v Repl.it.

Z obrázku 16 je zřejmé, že se kurz skládá z 5 kapitol. V těchto kapitolách postupně narůstá obtížnost (Podrobněji v podkapitole 3.1).

Druhou důležitou součástí hlavní stránky je navigační panel vlevo dole. Po kliknutí na první odkaz (zleva) se dostaneme na stránku indexu řešení. Index řešení vypadá podobně, jako index příkladů, liší se jiným nadpisem a vyměněnými barvami sloupců kapitol. Odkazy na této stránce vedou na stránku vyřešeného příkladu v Repl.it.

Druhý odkaz (Manuál) vede na PDF soubor, ve kterém je popsána struktura samotných příkladů v Repl.it. Tento manuál vznikl proto, aby se koncový uživatel lépe vyznal ve struktuře příkladů (kam je a kam není dovoleno zasahovat, co znamenají konkrétní oblasti příkladu a kde se vypisují).

Třetí a poslední odkaz vede na stránku „cheatsheetu“ (taháku). V něm jsou popsané základní znaky regulárních výrazů. Zde je nutno zmínit, že tento tahák nabízí pouze základní koncepty celé problematiky regulárních výrazů, není tedy samotný dostačující pro vyřešení složitějších příkladů v pozdějších kapitolách.

Na obrázku 17 je snímek první části cheatsheetu.

Kurz Regulárních výrazů - Cheatsheet

| Výraz/znak | Popis | Příklad |
|---|---|---|
| cokoliv | Zachytí jakýkoliv text | /cokoliv/ (zachytí "cokoliv") |
| ^ | Začátek řetězce (nebo řádku - flag "m") | /^abc/ (zachytí "abcneco", "abcregex", atd.) |
| $ | Konec řetězce (nebo řádku - flag "m") | /^.+konec$/ (zachytí "Tohle je konec.") |
| . | Jeden jakýkoliv znak | /reg.x/ (zachytí "regex", "regax", atd.) |
| \| | Operátor OR | /regex\|regax/ (zachytí "regex" i "regax") |
| \ | Escapování speciálního znaku | /\./ (zachytí ".") |
| + | Jeden nebo více (do nekonečna) | /\.+/ (zachytí "...") |
| * | Žádný nebo více (do nekonečna) | /regex\.*/ (zachytí "regex", "regex.", atd.) |
| \d | Číslo od 0 do 9 | /\d+/ (zachytí "123", "12345", atd.) |
| \w | Písmeno a až z, A až Z nebo číslo 0-9 | /\w+/ (zachytí "regx","reGEX","re12gex",atd.) |
| \t | Zachytí odtabování | /a\ta/ (zachytí "a a") |
| \D | Není číslo (obdoba [^0-9]) | /[\D]/ (zachytí cokoliv jiného, než číslo) |
| () | Skupina zachycení | /reg(ex\|ax)/ (zachytí "regex" a "regax")/ |
| ? | Jeden nebo žádný | /rege(x)?/ (zachytí "rege" |

Index příkladů | Manuál | Index řešení

Powered by 000webhost

Obrázek 17 – Snímek cheatsheetu

## 3.1: Struktura kapitol

Jak již bylo zmíněno v kapitole 2 (Struktura reg. výrazů), obtížnost narůstá postupně, tedy 1. kapitola je nejlehčí a 5. nejsložitější. V 1. až 4. kapitole se uživatel zabývá pouze regulárním výrazem samotným, v 5. kapitole se píšou už celé funkce a více se používají regex flagy.

V 1. kapitole se probírají naprosté základy regulárních výrazů (znaky „^“ ; „$“ ; „.“).

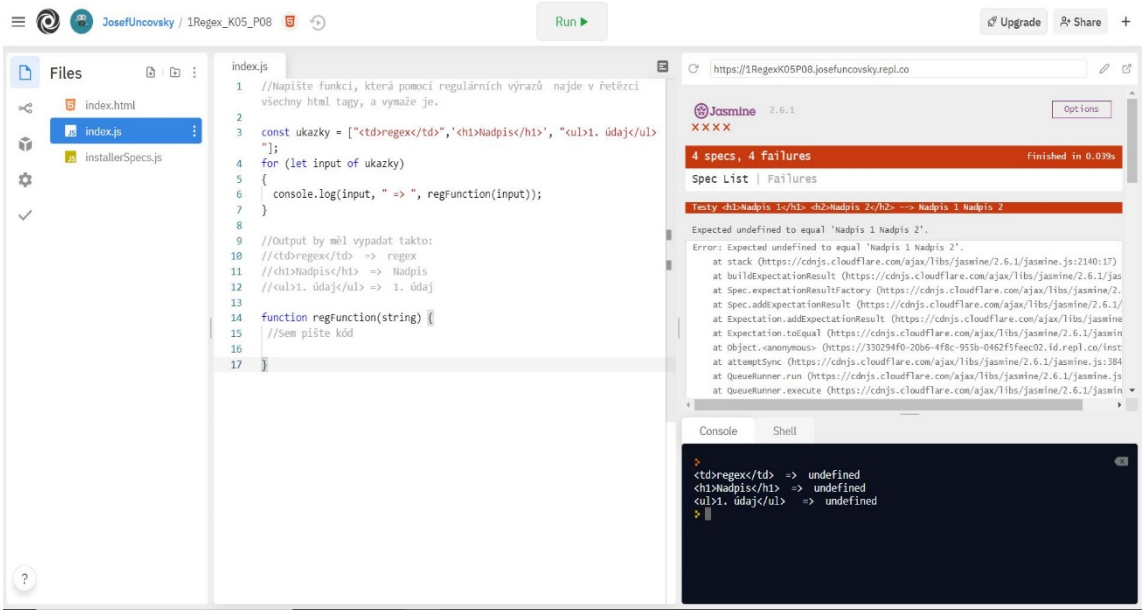
Ve 2. kapitole se k těmto znakům přidají znaky pro skupiny vyhledávaní ( „()“ ; „[]“ a „?“), přičemž řešení příkladů této kapitoly vyžaduje pochopení znaků probraných v 1. kapitole (Motiv, který se opakuje až do poslední kapitoly), protože úspěšné řešení příkladů zpravidla vyžaduje kombinaci těchto znaků.

Ve 3. kapitole se přidají další znaky (konkrétně znaky „=“ ; „{}“ ; „*“ ; „+“ a přidají se třídy znaků), a řešení příkladů zpravidla zahrnuje i znalosti z předchozích kapitol.

Ve 4. kapitole se probírají backslashe (zpětná lomítka) a jejich použití pro speciální znaky a velký důraz se klade na varianty řetězců, se kterými se řešení příkladů shoduje (např. řešení musí zachytit telefonní číslo ve tvaru „123 456 789“ i (123) 456 789 i (123).456-789, uživatel toto musí při řešení vzít v potaz a vytvořit výraz, který všechny tyto varianty zachytí.)

V 5. kapitole (jak již bylo zmíněno na začátku této kapitoly) se píšou celé funkce a mimo nových konceptů (regex flagy, syntaxe RegExp.$1-9) je pro úspěšné řešení důležité i ovládání a porozumění konceptům z předchozích kapitol.

## 3.2: Struktura příkladů



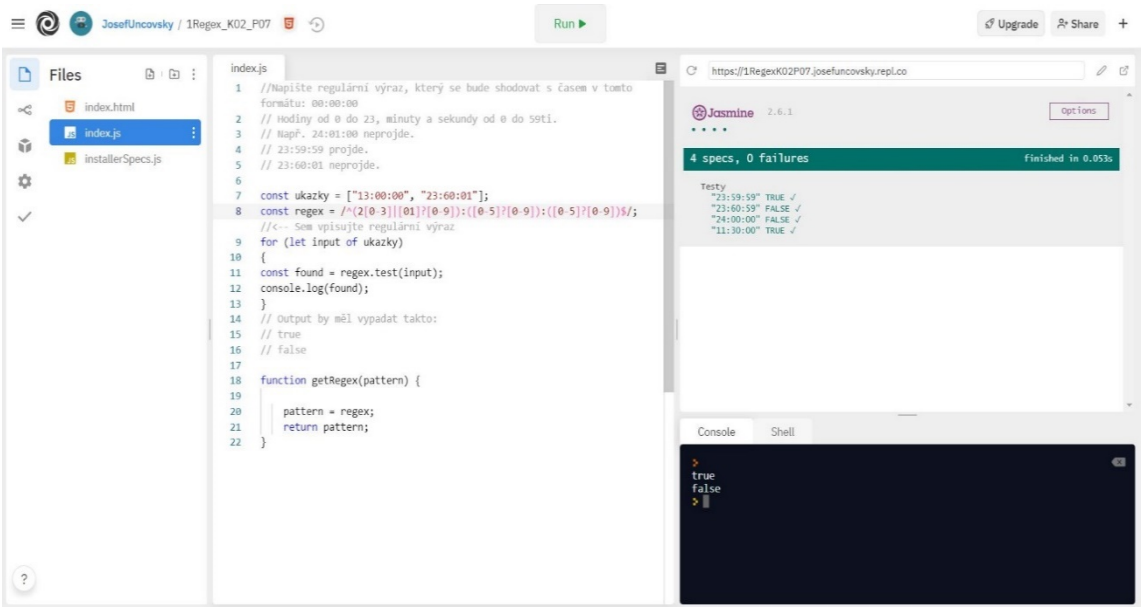
Obrázek 18 – Struktura příkladů 5. kapitoly v Repl.it

Na obrázku 18 vidíme stránku příkladu v Repl.it (Konkrétně se jedná o 8. příklad 5. kapitoly – HTML tagy). Všechna práce na příkladu se děje v souboru *index.js*. Toto je jediný soubor, se kterým řešitel příkladu pracuje (Všechny příklady jsou v tomto stejné – 3 soubory – *index.html* ; *index.js* ; *installerSpecs.js*).

Na prvních řádcích souboru (Toto pořadí je opět ve všech příkladech stejné) je v komentáři zadání, na 3. řádku pak začíná zkoušková oblast. Pod ní je v komentáři výstup, tak, jak by měl vzhledem k řetězcům v poli *ukázky* a k příkladu, vypadat. V nejspodnějších řádcích je pak hlavní funkce příkladu (v příkladech 5. kapitoly se do ní píše všechen kód, v příkladech ostatních kapitol se zapisuje jen regulární výraz do proměnné – jak je vidět na obrázku 19).

Zkoušková oblast se vypisuje v konzoli (Na obrázcích 18 a 19 oblast vpravo dole). Hlavní oblast se vypisuje přímo nad ní. Na obrázku 18 můžeme vidět, že řešení (jelikož je hlavní funkce prázdná) selhalo. Poznáme to podle 4 křížků (křížek označuje selhání) pod logem Jasmine, a také podle červeného motivu výpisu.

Pro řešitele je nejdůležitějším údajem selhávajícího řešení řádek/řádky pod druhým červeným řádkem, tedy řádek začínající slovy „Expected undefined“. Na tomto řádku je totiž uveden testcase hlavního příkladu, na kterém řešení selhává. Řešitelům se doporučuje tento testcase zkopírovat do pole *ukázky* a pracovat s ním i ve zkouškové oblasti.



Obrázek 19 – Struktura příkladů 1-4. kapitoly v Repl.it

Na obrázku 19 můžeme vidět příklad, jehož struktura odpovídá příkladům 1-4. kapitoly (Konkrétně se jedná o 7. příklad 2. kapitoly – Čas). Oproti příkladu na obrázku 18 si můžeme povšimnout, že řešení prošlo. Barevný motiv je zelený (na rozdíl od červené, pokud řešení selhává) a místo křížků jsou pod logem Jasmine zelené tečky (ty značí, že testcase je splněn). Pokud by např. druhý testcase neprošel, pod logem Jasmine by byly 3 zelené tečky a jeden křížek (v pořadí tečka – křížek – tečka – tečka).

Jak již bylo zmíněno, v příkladech 1-4. kapitoly je řešením pouze samotný regulární výraz, který se zapisuje do proměnné. Konkrétně se jedná o proměnnou *regex* (na obrázku 18 na 8. řádku).

## 3.3: Repl.it

Jak již bylo zmíněno v předchozích kapitolách, jako prostředí pro řešení samotných příkladů byla vybrána platforma *Repl.it*, což je online IDE (*Integrated Development Environment – Vývojové prostředí).* Tato platforma podporuje přes 50 programovacích a značkovacích jazyků a její rozhraní podporuje i většinu unit-testových frameworků, je tedy velmi flexibilní.

Pro samotný kurz byla tato platforma zvolena díky již zmíněné flexibilitě, neomezenému množství uživatelem vytvořených aplikací (každý příklad je samostatná webová aplikace a některá online vývojová prostředí mají limit např. 25 aplikací na uživatele) a také díky velice jednoduché implementaci unit-testového frameworku Jasmine do samotného rozhraní Repl.it.

Tato zmíněná implementace sestává z načtení zdrojových kódů Jasmine ze vzdáleného úložiště (to se v příkladech děje v souboru *index.html*) a vytvoření testů (soubor installerSpecs.js).

Testy se nakonfigurují tak, aby se buď testoval regulární výraz v proměnné *regex* oproti testovacímu řetězci (příklady kapitoly 1-4), nebo zdali vracená hodnota funkce *regFunction* v příkladu odpovídá testovacímu řetězci (příklady kapitoly 5).

Samotný framework Jasmine je velmi kompaktní a rychlý, takže se všechny testy provedou v řádu setin vteřiny.

Obrázek 20 – logo Repl.it

## 3.4: Jasmine

Jak již bylo zmíněno v předchozí kapitole, unit-testový framework Jasmine byl pro projekt zvolen ze dvou důvodů: Rychlost provedení testů a velice jednoduchá implementace v prostředí Repl.it. Oproti jiným testovacím frameworkům pro JavaScript (např. Mocha nebo Jest) má Jasmine ale i jiné výhody: samotná syntaxe Jasmine je velice jednoduchá. [9] Na obrázku 21 je ilustrován test jednoho z příkladů kurzu regulárních výrazů (Konkrétně se jedná o 5. příklad 4. kapitoly – URL)

```
describe('Test', () => {
    let pattern;
    it('http://localhost:8500/', () => {
        expect('http://localhost:8500/').toMatch(getRegex(pattern));

    })
});
```

Obrázek 21 – Příklad syntaxe Jasmine

Samotné vytvoření sady testů se děje na prvním řádku. Na druhém řádku se ze souboru *index.js* načte proměnná *regex*, která se pro přehlednost v hlavní funkci příkladu přejmenuje na *pattern* (obrázek 19) a na 3. až 5. řádku se vytvoří samotný testcase. Všechny testy příkladů 1. až 4. kapitoly mají stejnou strukturu, jako na obrázku 21. Každý jednotlivý testcase obsahuje podmínku – pokud se testovací řetězec shoduje s regulárním výrazem, testcase projde. Pokud ne, test ohlásí nesplnění podmínky a vypíše řetězec, se kterým se řešení neshoduje (jak již bylo zmíněno v podkapitole 3.2).

# 4: Závěr

Cílem mého projektu bylo vytvořit celistvý kurz regulárních výrazů.

Nejprve bylo nutné vybrat programovací jazyk, ve kterém se příklady později budou řešit. Poté bylo vybráno prostředí pro řešení příkladů a Unit-testový framework, který řešení vyhodnocuje. Následně byla do hloubky prostudována celá problematika regulárních výrazů (Za pomocí odborných publikací a webových zdrojů), problematika práce s Unit-testovým frameworkem Jasmine a následně začala tvorba samotných příkladů.

Poté byl vytvořen rozcestník celého kurzu, který byl později obohacen o cheatsheet a manuál, ve kterém je popsáno, jak se s příklady/prostředím pracuje (Tento manuál obsahuje také ukázková videa). Souběžně s tvorbou rozcestníku vznikal i rozcestník s řešeními jednotlivých příkladů.

Vlivem nepříznivých časových podmínek se výsledek mé práce bohužel nedostal ke studentům 3. ročníku. Jejich zpětná vazba by pro projekt byla jistě hodnotná.

# 5: Bibliografie a citace


[1] Jan Goyvaerts, Steven Levithan, Autoři, *Regular expressions cookbook*. [Kniha] 2009. ISBN – 9780596520687 [Přístup 12. září 2020]

[2] Wikipedia, the free encyklopedia, „Grep“ [Online]. Available: https://en.wikipedia.org/wiki/Grep[Přístup 14. září 2020]

[3] Wikipedia, the free encyklopedia, „Regular expression“ [Online]. Available: https://en.wikipedia.org/wiki/Regular_expression[Přístup 14. září 2020]

[4] MDN web docs, „RegExp – JavaScript“,[Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp [Přístup 11. říjen 2020]

[5] MDN web docs, „String.prototype.matchAll() - JavaScript“, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/matchAll [Přístup 18. říjen 2020]

[6] MDN web docs, „RegExp.prototype.test() - JavaScript“,[Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/test [Přístup 25. říjen 2020]

[7] MDN web docs, „String.prototype.match() - JavaScript“, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match [Přístup 4. listopad 2020]

[8] javascript info, „regexp-introduction“, [Online]. Available: https://javascript.info/regexp-introduction [Přístup 11. listopad 2020]

[9] Jasmine.github.io, „Custom_matcher“, [Online]. Available: https://jasmine.github.io/tutorials/custom_matcher [Přístup 18. listopad 2020]

# 6: Seznam obrázků