

Vizualizace regulárních výrazů

Regular Expression Visualization

Dominik Kundra

Bakalářská práce

Vedoucí práce: Ing. Jakub Beránek

Ostrava, 2024

Zadání bakalářské práce

Student:

Dominik Kundra

Studijní program:

B0613A140014 Informatika

Téma:

Vizualizace regulárních výrazů
Regular Expression Visualization

Jazyk vypracování:

čeština

Zásady pro vypracování:

Cílem práce je vytvořit nástroj sloužící pro vizualizaci a ladění regulárních výrazů. Nástroj by měl být schopný zpracovat zvolený regulární výraz, sestavit plán vykonávání daného výrazu dle zvolené implementace a poté umožnit programátorovi interaktivně krokově provádění regulárního výrazu. Nástroj by měl být vytvořen jako rozšíření do vývojového prostředí (např. do Visual Studio Code), aby šel jednoduše použít při vývoji programů. Výsledná aplikace by měla být řádně zdokumentována a při jejím vývoji by měl být využit verzovací systém (např. git).

1. Analyzujte a popište možnosti implementace regulárních výrazů.
2. Navrhnete architekturu rozšíření do vývojového prostředí, které bude schopné analyzovat regulární výrazy ze zvoleného zdrojového kódu.
3. Naimplementujte nástroj pro vizualizaci regulárního výrazu a integrujte jej do vývojového prostředí.
4. Otestujte vizualizaci nástroje na regulárních výrazech z reálných projektů.

Seznam doporučené odborné literatury:

- [1] FRIELD, Jeffrey. Mastering Regular Expressions 3rd Edition. 2006. O'Reilly Media. ISBN: 978-0596528126
- [2] SORVA, Juha. Visual program simulation in introductory programming education. Espoo: Aalto Univ. School of Science, 2012. ISBN 9789526046266. Dostupný také z WWW: <http://doi.acm.org/10.1145/2445196.2445368>.
- [3] VANDERKAM, Dan. Effective TypeScript : 62 Specific Ways to Improve Your TypeScript. O'Reilly Media, 2019. ISBN 978-1492053699

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jakub Beránek**

Datum zadání: 01.09.2023

Datum odevzdání: 30.04.2024

Garant studijního programu: doc. Mgr. Miloš Kudělka, Ph.D.

V IS EDISON zadáno: 09.11.2023 15:43:31

Abstrakt

Tato práce se věnuje vizualizaci regulárních výrazů, slouží pro pochopení jejich fungování a k potenciálnímu odhalení chyb při jejich ladění. První část práce popisuje principy fungování regulárních výrazů, historii a jejich základní vzory. Druhá část srovnává existující nástroje, jejich možnosti a inspirace pro tuto práci. Třetí část probírá specifikaci požadavků, zvolený návrh a použité technologie aplikace. Dále se detailněji zabývá implementací samotné aplikace, konkrétně zpracováním regulárních výrazů, vizualizací a rozšířením pro vývojové prostředí Visual Studio Code.

Klíčová slova

vizualizace regulárních výrazů; vývojové prostředí Visual Studio Code

Abstract

This thesis focuses on visualization of regular expressions, serving a purpose for understanding their functioning and for potential error detection while debugging them. First part of thesis describes functioning principles of regular expressions, history and their basic patterns. Second part compares existing tools, their options and inspirations for this thesis. Third part discusses specification of requirements, selected design and used technologies in application. Furthermore this part more focuses on implementation of the application alone, specifically about regular expression evaluation, visualisation and extension for development environment Visual Studio Code.

Keywords

visualization of regular expressions; development environment Visual Studio Code

Poděkování

Chtěl bych poděkovat svému vedoucímu práce, Ing. Jakubovi Beránkovi, za ochotu, pomoc a za rychlou zpětnou vazbu při realizaci této práce.

Obsah

Seznam použitých symbolů a zkratk	7
Seznam obrázků	8
Seznam zdrojových kódů	9
Seznam tabulek	10
1 Úvod	11
2 Principy a historie regulárních výrazů	13
2.1 Formální jazyk	13
2.2 Konečný automat	13
2.3 Bezkontextová gramatika	15
2.4 Vznik, implementace a vzory	15
3 Existující vizualizační nástroje	19
3.1 Regex101	19
3.2 RegExr	20
3.3 Debuggex	20
3.4 Visual Studio Code přístupy	20
4 Architektura aplikace	21
4.1 Specifikace požadavků	21
4.2 Návrh aplikace	21
4.3 Použité technologie	23
5 Knihovna pro práci s regulárními výrazy	25
5.1 Architektura	25
5.2 Parsování regulárních výrazů	27
5.3 Vyhledání pomocí regulárního výrazu	29

6	Vizualizační knihovna a rozšíření	34
6.1	Návrh vizualizační části	34
6.2	Implementace textových editorů	34
6.3	Vizualizace průchodu	37
6.4	Uživatelské rozhraní	38
6.5	VSCode rozšíření	40
6.6	Sestavení a spuštění projektu	41
7	Zhodnocení a testování výsledků	43
8	Závěr	45
	Literatura	46
	Přílohy	46
A	Adresářová struktura projektu	47
B	Testovací data	48

Seznam použitých zkratk a symbolů

AST	– Abstraktní syntaktický strom
NKA	– Nedeterministický konečný automat
DKA	– Deterministický konečný automat
TS	– TypeScript
JS	– JavaScript
Regex	– Regulární výraz (Regular expression)
API	– Aplikační rozhraní (Application Programming Interface)
HTML	– HyperText Markup Language
CSS	– Cascading Style Sheets
VSCoDe	– Visual Studio Code
PCRE	– Perl-Compatible Regular Expressions

Seznam obrázků

2.1	Příklad deterministického automatu přijímající slova obsahující písmena z abecedy {a, b} končící písmenem a	14
2.2	Příklad nedeterministického automatu ekvivalentního k předchozímu deterministickému	15
2.3	Převedený prázdný výraz ϵ	16
2.4	Převedený výraz a	16
2.5	Převedený výraz s t	17
2.6	Příklad regulárního výrazu	18
4.1	Struktura knihoven aplikace	22
5.1	Třídní diagram části knihovny pro práci s regulárními výrazy	26
5.2	Příklad výsledné struktury regulárního výrazu a^+	29
5.3	NKA pro popis počítání iterací a prevenci nekonečných cyklů	31
6.1	Třídní diagram textových editorů	35
6.2	Úvodní uživatelské rozhraní	39
6.3	Uživatelské rozhraní debuggeru	40

Seznam zdrojových kódů

5.1	Jednoduché pravidlo gramatiky	27
5.2	Výběry neterminálů pro některé vzory regulárních výrazů	28
5.3	Vložení stavu do zásobníku	30
5.4	Vyvolání backtrackingu, pokud neexistují další přechody ze současného stavu	30

Seznam tabulek

7.1	Výsledky testování výkonu zpracování regulárních výrazů	43
-----	---	----

Kapitola 1

Úvod

Vyhledávání v textu patří mezi základní problémy, se kterými se velmi pravděpodobně potká skoro každý programátor. Tento problém se dá řešit mnoha způsoby, avšak ne všechna řešení lze použít univerzálně a každý ze způsobů má své výhody a nevýhody. Jedním ze přístupů je využití **regulárních výrazů** (regexů). Jedná se o sekvenci znaků, která nám umožňuje popsat nějakou množinu slov. Nejčastěji je jejich výsledná forma v podobě **konečného automatu**. Konečné automaty jsou blíže vysvětleny v sekci 2.2. Téměř každý programovací jazyk v dnešní době obsahuje regulární výrazy, ale jejich implementace se mohou lišit.

Cílem této práce je naimplementovat nástroj, který bude schopný zpracovávat a procházet regulární výrazy. Následně lze vizualizovat tyto průchody, a to jako součást rozšíření ve zvoleném vývojovém prostředí.

Při vývoji programů se programátor často setkává s regulárními výrazy, jedná se totiž o poměrně rychlý způsob vyhledávání v textu. Můžeme se s nimi setkat v podstatě skoro ve všech částech softwaru, např. validace formulářů, vyhledávání v textu, nebo například v příkazovém řádku. Avšak v těchto výrazech je někdy poměrně složité se vyznat, jelikož neumožňují téměř žádné formátování. Taktéž mohou být pro mnoho lidí matoucí, či nepřehledné. Z tohoto důvodu je vhodné mít nástroj, který potencionálně usnadní práci programátorům, tak aby si mohli zobrazit průchod zadaným výrazem. Dále pro někoho, kdo například vidí tyto výrazy poprvé v životě, může být snazší jim porozumět, existuje-li možnost zobrazit princip jejich fungování v jednotlivých krocích. Sice již existují řešení tohoto problému, a to v různých formách [1, 2, 3], ale pro zvolené prostředí Visual Studio Code (VSCode) mnoho přístupů neexistuje. Tato situace je motivací zabývat se problémem a pokusit se nabídnout originální řešení v daném směru, které by mohlo být přínosem pro ostatní.

Implementace těchto výrazů bývá nejčastěji formou konečných automatů, jedná se o poměrně výkonné řešení. Aby bylo možné tohoto dosáhnout musí být převedena jejich textová forma na strukturu konečného automatu. Toho může být dosaženo například využitím bezkontextové gramatiky, nebo implementací vlastního parseru. Bezkontextová gramatika je formální jazyk, který pomocí pravidel definuje nějaký jazyk. Parser slouží pro analýzu a zpracování textového řetězce

a je výsledkem překladu bezkontextové gramatiky do kódové reprezentace. Později v kapitole 4 je vysvětleno, který ze způsobů byl zvolen a důvod této volby.

Vizualizaci regulárních výrazů je možné chápat několika způsoby, lze si ji např. představit jako zobrazení ekvivalentního konečného automatu. Další možný přístup je mapování stavů automatu do původní textové podoby. Druhý přístup jsem zvolil pro tuto práci, a to ve smyslu **ladícího nástroje (debugger)**. Debugger v tomto případě funguje jako historie jednotlivých kroků průchodu zadaným výrazem. Tento průchod se také nazývá jako krokování.

Regulární výrazy pocházejí z **teoretické informatiky**¹, byly nadefinovány roku 1956, ale k jejich využití v počítačích se dostalo až v roce 1968 v operačním systému **UNIX**. Od své původní formy se dnes ve svém základu téměř neliší, ale často již obsahují složitější funkcionality a rozšířenou syntaxi. Jedno z jejich nejznámějších využití je v příkazovém řádku v linuxových operačních systémech, původně v UNIXu, a to pod názvem **g/re/p** neboli **grep** „Global search for Regular Expression and Print matching lines“[4].

V kapitole 2 jsou podrobněji popsány pojmy z teoretické informatiky, dále implementace regulárních výrazů, jejich vzory a vznik. Popis existujících nástrojů a přístupů řešení se nachází v kapitole 3. Následně kapitola 4 popisuje specifikaci požadavků, návrh aplikace a použité technologie při vývoji aplikace. Kapitola 5 se zabývá implementací knihovny pro zpracování regulárních výrazů. Ta je pak využita v samotné vizualizaci, která je blíže popsána v kapitole 6, společně s rozšířením pro *Visual Studio Code*. Zhodnocení a testování výsledků je blíže popsáno v kapitole 7. Poslední kapitola 8 shrnuje obsah práce, dosažené výsledky a možné rozšíření aplikace do budoucna.

¹Vědní obor na pomezí mezi informatikou a matematikou.

Kapitola 2

Principy a historie regulárních výrazů

Tato kapitola se zabývá definicí regulárních výrazů, jejich fungováním a jak se jednotlivé implementace mohou lišit. Součástí jejich implementace provází několik pojmů z teoretické informatiky, odkud pocházejí. Hlavně se jedná o **Konečné automaty** a **Thompsonovo sestavení**.

2.1 Formální jazyk

Formální jazyk je libovolná množina konečných slov nad určitou abecedou [5]. Slova chápeme jako řetězce znaků, která jsou přijímaná zadaným jazykem. Délka slov musí být sice konečná, ale množina těchto slov může být nekonečná. Tyto jazyky mohou být například definovány regulárními výrazy, formální gramatikou nebo konečnými automaty.

2.2 Konečný automat

S regulárními výrazy se často pojí konečné automaty, jedná se o další oblast z teoretické informatiky. Tato práce implementuje regulární výrazy právě ve formě konečných automatů. Zjednodušeně se dá říct, že konečný automat je výpočetní model jednoduchého počítače, který má určitý počet stavů a přechodů [6].

Definice

Konečné automaty se dělí na **deterministické** a **nedeterministické**, zkráceně **DKA** (deterministický konečný automat) a **NKA** (nedeterministický konečný automat). DKA mohou mít v daném stavu pro každý znak abecedy **právě jeden** přechod, zatímco NKA umožňují více stejných přechodů z daného stavu. NKA také mohou obsahovat tzv. prázdný znak často označovaný řeckým písmenem epsilon ϵ . Prázdné znaky slouží pro změnu stavu bez změny aktuální pozice ve hledaném slově. Na závěr lze podotknout, že každý NKA je možné převést na ekvivalentní DKA.

Deterministický konečný automat je pětice $(Q, \Sigma, \delta, q_0, F)[7]$:

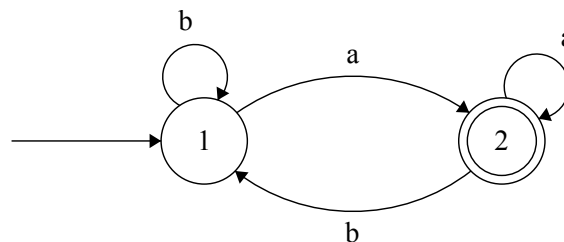
- Q — konečná množina stavů.
- Σ — konečná vstupní abeceda.
- $\delta : Q \times \Sigma \rightarrow Q$ — přechodová funkce.
- $q_0 \in Q$ — počáteční stav.
- $F \subseteq Q$ — množina konečných stavů.

Nedeterministický konečný automat je pětice $(Q, \Sigma, \Delta, S, F)[7]$:

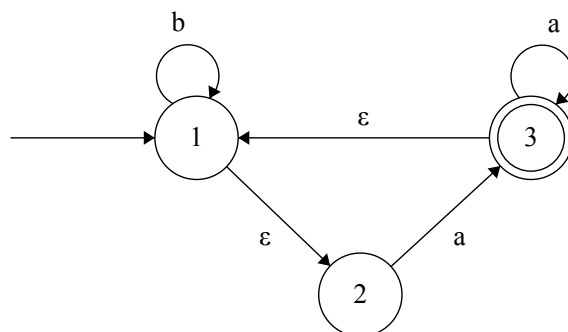
- Q — konečná množina stavů.
- Σ — konečná vstupní abeceda.
- $\Delta : Q \times \Sigma \rightarrow 2^Q$ — přechodová funkce, kde stav a vstupní symbol určuje množinu možných následujících stavů.
- $S \subseteq Q$ — množina počátečních stavů.
- $F \subseteq Q$ — množina konečných stavů.

Vizualizace

Stavy jsou typicky zakreslovány jako kružnice. Konečné stavy se označují jako kružnice s dvojitou čarou. Počáteční stavy jsou označovány jako stav, do kterého vede šipka, která ale nevychází z jiného stavu. Přechody jsou znázorněny jako šipky vedoucí z jednoho stavu do druhého a jsou označeny přechodovým symbolem. Pro upřesnění, přechod může odkazovat na stejný stav ze kterého vychází. Tyto přechody nám říkají, že pokud chceme přejít z jednoho stavu do druhého, tak se musíme v přijímaném slově posunout o přechodový symbol. Pokud to není možné, tak nelze tímto přechodem přejít do tohoto stavu. Pro ukázkou lze porovnat dva ekvivalentní konečné automaty, NKA na obrázku 2.2 a DKA na obrázku 2.1.



Obrázek 2.1: Příklad deterministického automatu přijímající slova obsahující písmena z abecedy $\{a, b\}$ končící písmenem a



Obrázek 2.2: Příklad nedeterministického automatu ekvivalentního k předchozímu deterministickému

2.3 Bezkontextová gramatika

Součástí této práce je i využití Bezkontextové gramatiky, pro nadefinování syntaxe regulárních výrazů.

Bezkontextová gramatika je jedna další z možných definic formálních jazyků. Je určena konečnou množinou **neterminálních symbolů** (proměnných), konečnou množinou **terminálních symbolů**, která nesmí mít žádné prvky společné s předchozí množinou. Dále je součástí **počáteční neterminál**, s konečnou množinou **přepisových pravidel**[5].

Pro příklad může sloužit výraz $A \rightarrow \beta$, kde A je neterminál a β je řetězec složený z terminálů a/nebo neterminálů. Dále šipka indikuje **přepsání** tzn. levá strana se přepisuje na stranu pravou. Konečný řetězec generovaný danou gramatikou, je pouze tvořen terminálními symboly. Aby mohl být řetězec přijímán zadanou gramatikou, musí ho být schopná vygenerovat.

2.4 Vznik, implementace a vzory

Regulární výrazy byly poprvé nadefinovány Americkým matematikem **Stephan Cole Kleenem**, jako regulární jazyky. Dále se aplikovaly v teoretické informatice, jako podkategorie **teorie automatů** a součást **formálních jazyků**. Ačkoliv byly nadefinovány začátkem padesátých let, tak jeho využití v počítačích nastalo až na konci šedesátých let a to v jednom z nejznámějších operačních systémů UNIX.

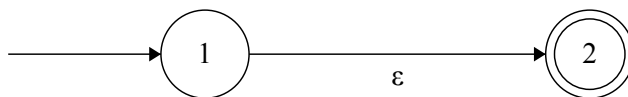
Thompsonovo sestrojení

První kdo navrhl implementaci používanou v počítačích byl **Ken Thompson**. Principem byl převod regulárního výrazu na NKA. Tato metoda se často používá doposud, v podobné či nezměněné podobě. Algoritmus se pojmenoval **Thompson's construction** (Thompsonovo sestrojení), který převádí textovou reprezentaci výrazu na ekvivalentní nedeterministický automat. Toto sestrojení je využito v této práci a blíže jej popisuje následující část textu.

NKA se běžně využívá, jelikož je poměrně jednoduchý na implementaci. Také oproti DKA využívá **zpětného krokování** (backtracking) a povoluje složitější operace jako je *rozhlédnutí se kolem sebe* (look-around). Backtracking je důležitý pro NKA, jelikož neexistuje jednoznačná cesta vyhodnocení. To znamená, že pokud je NKA ve stavu, ze kterého nelze pokračovat dále, tak je potřeba se vrátit do předchozího stavu. DKA mají výhodu, že jsou rychlejší, ale jsou typicky mnohem větší než jejich ekvivalentní NKA a neumožňují lehce implementovat některé složitější operace. Také nepotřebují zpětné krokování, jelikož jejich cesta je deterministická, tzn. existuje vždy jen jedna cesta pro hledané slovo. Dnes se ale často využívá kombinace DKA i NKA, kdy DKA se využije pro rychlé vyhledání daného slova a pokud bylo slovo nalezeno, tak se použije NKA pro jejich rozšířené možnosti.

Výsledný NKA po Thompsonově sestrojení má právě jeden vstupní a výstupní stav. Thompsonovo sestrojení dále definuje několik následujících pravidel.

Prázdný výraz ϵ , je převedený na vstupní stav, přechod ϵ a konečný stav. Výsledný konečný automat je na obrázku 2.3.



Obrázek 2.3: Převedený prázdný výraz ϵ

Výraz a , je převedený podobně jako prázdný výraz, ale s rozdílem přechodu a místo ϵ . Konečný automat, který tímto převodem vznikne je ukázán na obrázku 2.4.



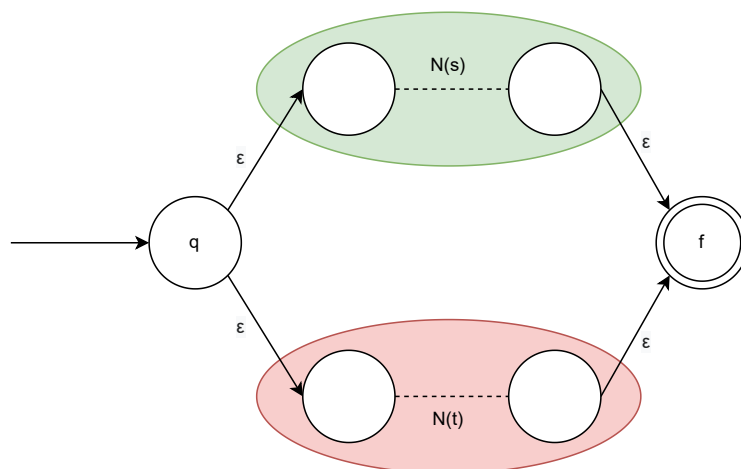
Obrázek 2.4: Převedený výraz a

Pro zadaný výraz $s|t$ (varianta), kde s je levá strana varianty a t je pravá strana varianty, platí, že ze stavu q (počáteční stav) vedou dva přechody ϵ , na počáteční stavy variant s a t . Z těchto počátečních stavů dále pokračuje sekvence stavů $N(s)$ pro s a $N(t)$ pro t . Konce variant s a t mají každé jediný přechod ϵ na konečný stav f . Na obrázku 2.5 je znázorněný výsledný NKA, kde skupina stavů v zelené části je s a červená skupina je t .

Další pravidla pro sestrojení lze například najít v následujících člancích [8, 9]. Některé pravidla jsou v této práci upravená, ale fungují na stejném principu.

Základní vzory regulárních výrazů

V předchozích sekcích již byly popsány základní konstrukce týkající se regulárních výrazů. Tato sekce se zabývá jejich základními vzory, neboli jejich formou zápisu a syntaxí. Popis následujících vzorů vychází ze syntaxe regulárních výrazů pro jazyk JavaScript.



Obrázek 2.5: Převedený výraz $s|t$

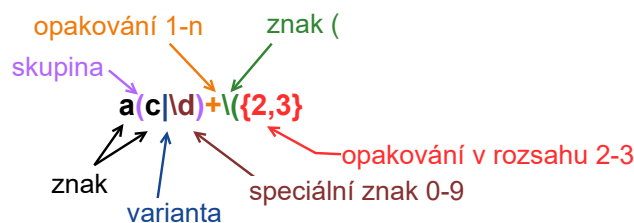
Za nejjednodušší výraz lze považovat prázdný výraz, také označovaný jako ϵ . Tento výraz dokáže přijímat slova délky 0, resp. prázdná slova. Výrazy mohou obsahovat **téměř** libovolný znak, který bude přijímat slova s tímto znakem. Avšak nemohou být použity znaky, které jsou rezervované, neboli jsou součástí syntaxe regulárních výrazů. Chceme-li použít tyto znaky, je potřeba použít zpětné lomítko \backslash . Takové spojení znaku a zpětného lomítka se pak anglicky nazývá **escaped character**. Také existují znaky, které nejsou součástí rezervovaných znaků, ale lze před nimi použít zpětné lomítko. Funkcionalita těchto znaků se následně mění, např. pokud použijeme zpětné lomítko před znakem d , tak to ve výrazu značí přijmutí čísla od 0 do 9.

Iterace je možnost jak lze opakovaně provádět nějaký vzor. Například lze iterovat znak, skupinu a další konstrukce. Nelze však opakovat jakýkoliv vzor. Prvním typem iterace je *****, známa jako **Kleene star**. Tento druh iterace může mít počet opakování od **0** do **n**. Dále existují další 2 typy iterací, a to je iterace v rozmezí $1 - n$ označována znakem $+$ a *iterace v rozmezí*, která se značí $\{od, do\}$. V regulárních výrazech se vždy vzor opakování vyskytuje za konstrukcí, které se má opakovaně provádět.

Operace *varianta* je dalším základním vzorem pro regulární výrazy. Jedná se o výběr mezi pravou a levou stranou. Oddělovacím znakem je typicky $|$ podobně jako bitová operace *OR* v mnoha programovacích jazycích.

Dalšími základními konstrukty jsou například skupiny, které jsou obaleny v jednoduchých závorkách. Ty slouží k rozdělení částí regulárních výrazů, které jsou po dokončení vyhledávání přístupné jako oddělené části vyhledání.

Na obrázku 2.6 lze vidět příklad regulárního výrazu, ve kterém jsou použity a popsány některé ze zmíněných vzorů.



Obrázek 2.6: Příklad regulárního výrazu

Implementace v programovacích jazycích

V dnešní době mají v podstatě skoro všechny programovací jazyky nějakou formou implementované regulární výrazy. Implementace v programovacích jazycích se často liší svou syntaxí a obsáhlostí, ale jejich základ bývá stejný. Může se tak stát to, že funkcionality podporovaná jedním jazykem není podporována druhým. Taktéž oproti původním regulárním výrazům, dnešní implementace obsahují mnohdy složitější koncepce, jako je look-around nebo například rekurze. Někdy jazyky sice sdílí stejné konstrukce, ale mohou se lišit syntaxí.

Look-around je již celkem pokročilá funkcionality, jejímž principem je takzvané nezachytávání znaků při zpracovávání. Typicky je dělíme podle směru, a to na *dopředné* a *zpětné*. Pak je dělíme podle podmínění, a to na *kladné* a *negativní*. Pokud máme kladné podmínění, **musí** uzavřený výraz být splněný a pokud máme záporné, tak **nesmí** být splněný. V původní formě regulárních výrazů tato funkce neexistovala.

Mnohdy je potřeba nalezený řetězec rozdělit do skupin. Tuto možnost dnešní implementace také umožňují. Chceme-li zdůraznit, že zadaný podvýraz je skupinou, obalíme ho do závorek. Tato vlastnost je důležitá, jelikož není potřeba v již nalezeném řetězci hledat další podřetězce pomocí dalšího výrazu. Skupiny se dělí na zachytávající (capturing), pojmenované (named) a nezachytávající (non-capturing). Pojmenované patří pod zachytávající, akorát jsou identifikovány pomocí názvu místo indexu. Obě skupiny zůstávají zachycené po dokončeném vyhledávání. Nezachytávající skupiny slouží čistě pro regulární výrazy, například při opakování části výrazu, ale ve výsledku se již nenachází.

Asi nejobsáhlejší implementací je *PCRE* (Perl-Compatible Regular Expressions) a *PCRE2*. Tento standard pochází z jazyka Perl, ale také je například součástí jazyka PHP. Nachází se zde již poměrně složité vzory, jako jsou podmínky nebo rekurze.

Kapitola 3

Existující vizualizační nástroje

Pro vizualizaci regulárních výrazů existuje několik přístupů realizace tohoto problému. Avšak tato řešení se často liší a neexistuje jednotný způsob, který by implementoval všechny nástroje jednotně. Jednou z forem zobrazení je pomocí ladícího nástroje, neboli debuggeru. Další formou může být například zobrazení výsledného konečného automatu a případná vizualizace průchodu tímto automatem.

3.1 Regex101

Jedním z nejznámějších nástrojů pro vizualizaci regulárních výrazů je webová stránka *Regex101*². Ta využívá principu debuggeru, kde se průchody výrazem mapují do původního výrazu. Nachází se zde dvě textová pole, do kterých uživatel může psát regulární výrazy a text pro vyhledávání. Tato stránka také podporuje syntaxi regulárních výrazů různých programovacích jazyků. Jejich debugger pouze podporuje standardy *PCRE* a *PCRE2*.

Velkou výhodou tohoto nástroje je možnost zobrazit si seznam všech vzorů pro daný jazyk. Pokud se někdo poprvé seznamuje s těmito výrazy, může využít tohoto seznamu pro jejich rychlejší pochopení. Dále tento nástroj obsahuje mimo jiné zvýraznění částí textu, kde zadaný výraz úspěšně dokázal vyhledat shody. Také jsou zvýrazněny některé syntaktické prvky zadaného regulárního výrazu, jako jsou například skupiny. Mezi poslední funkcionality, kterými tento nástroj disponuje, je zobrazení abstraktního syntaktického stromu (AST) pro zadaný výraz. Tato AST struktura také obsahuje, popis jednotlivých částí výrazu.

Tento nástroj byl ve výsledku mou velkou inspirací pro tuto aplikaci. Jelikož se jedná o obsáhlou aplikaci, není možné v této práci naimplementovat stejné množství funkcionalit. Proto jsem se rozhodl použít alespoň některé z nich.

²<https://regex101.com>

3.2 RegExr

*RegExr*³ je dalším webovým nástrojem, který lze použít pro vizualizaci regulárních výrazů. Tato stránka již nedisponuje debuggerem. Spíše využívá jednoduššího zvýrazňování částí vyhledaného textu. Podobně jako *Regex101* se zde nachází zobrazení AST s popisem syntaxe.

Zajímavou částí této aplikace je možnost použití testovacích řetězců. V aplikaci se lze přepnout do sekce pro psaní vlastních testů. Uživatel zde může zadávat textové řetězce, ve kterých následně proběhne hledání pomocí zadaného výrazu. Pokud uživatel často mění zadaný regulární výraz, tak si může zkontrolovat, zda všechny napsané testy správně proběhnou.

3.3 Debuggex

Dalším přístupem pro vizualizaci je *Debuggex*⁴. Na rozdíl od zmíněných přístupů Debuggex využívá vizualizace nedeterministického konečného automatu. V NKA se následně nachází kurzor, který signalizuje aktuální pozici procházeného výrazu. Tato aplikace také zobrazuje pozici v původním regulárním výrazu, ale není tolik detailní jako v případě *Regex101*. Aktuální podpora jazyků je JavaScript, Python a PHP.

3.4 Visual Studio Code přístupy

Pro vývojové prostředí Visual Studio Code existuje několik nástrojů. Nejstahovanějším nástrojem je *Regex Previewer*⁵. Ten funguje na principu testovacího okna, ve kterém je zadaný text pro vyhledávání. Pokud se v kódu nachází regulární výraz, tak se ve vedlejším okně zvýrazní části textu, kde vyhledávání proběhlo úspěšně.

Mezi další nástroje, které existují pro zvolené prostředí, stojí za zmínku například *Visual Regex*⁶ společně s *Regexper unofficial*⁷. Tyto nástroje vygenerují obrázek NKA, který následně uživateli zobrazí. Nedostatkem těchto nástrojů je poměrně malá uživatelská interaktivita a flexibilita.

Existují další přístupy řešení pro VSCode. V případě mé aplikace jsem podobný přístup pro dané vývojové prostředí nenašel.

³<https://regexr.com>

⁴<https://debuggex.com>

⁵<https://github.com/chrmarti/vscode-regex>

⁶<https://github.com/roberthgnz/visual-regex>

⁷<https://github.com/APerricone/vscode-regexper-unofficial>

Kapitola 4

Architektura aplikace

4.1 Specifikace požadavků

Aplikace by měla umožňovat vizualizaci a ladění regulárních výrazů. Ladicí nástroj by měl být integrovanou součástí zvoleného vývojového prostředí. Vizualizace stavů průchodu regulárním výrazem by měla být ve formě historie s intuitivním a interaktivním ovládáním. Program by měl umět zpracovávat nejzákladnější a běžně používané vzory regulárních výrazů, tak aby mohl být v praxi použitelný na reálných příkladech. Dále by měla aplikace obsahovat následující prvky:

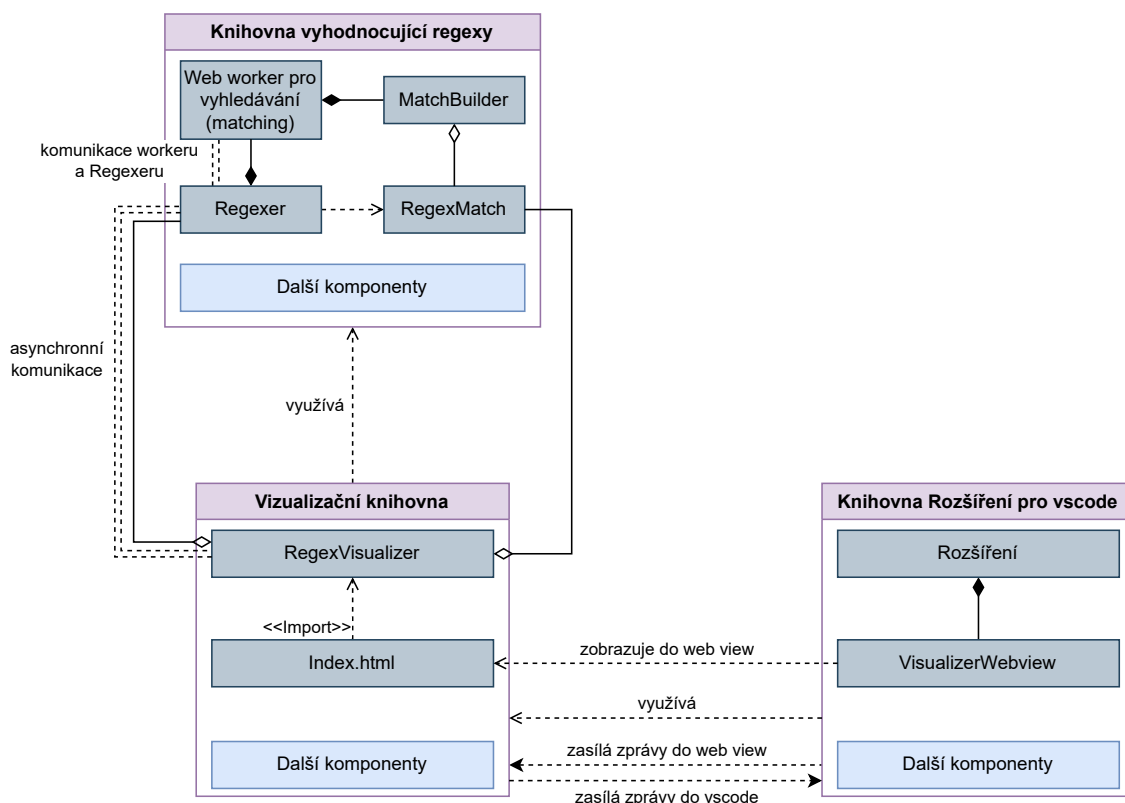
- Lehká budoucí rozšiřitelnost pro další možné funkcionality, které lze naimplementovat.
- Vizualizační část aplikace by měla obsahovat možnost zadávání regulárních výrazů a textu pro následné vyhledávání.
- Možnost spuštění mimo vývojové prostředí Visual Studio Code.
- Ladicí nástroj pro vizualizaci stavů.
- Implementaci základních vzorů regulárních výrazů.
- Možnost analýzy regulárních výrazů ze zdrojových kódů.

4.2 Návrh aplikace

Aplikace je členěná na tři základní knihovny, což je zřejmé na obrázku 4.1. Každá část má vlastní účel a jsou od sebe navzájem izolovány. První knihovna **vyhodnocuje regulární výrazy**, přesněji slouží pro zpracovávání a vyhodnocování regulárních výrazů a pro vytváření historie stavů vyhledávání. Jako jediná může existovat plně nezávisle bez ostatních knihoven, jelikož neobsahuje závislost na žádnou z nich. Druhá knihovna je **vizualizační**, která slouží pro samotné zobrazení zpracovávaných regulárních výrazů. Jedná se o komponentu, která může být spuštěná mimo rozšíření VSCode,

například ve webovém prohlížeči. Tato knihovna závisí na knihovně pro zpracovávání regulárních výrazů, ale na VSCode rozšíření nemá přímou závislost. Poslední částí je samotné **rozšíření**, které se stará o komunikaci s aplikačním rozhraní (API) VSCode a o řízení všeho týkajícího se rozšíření. Tato část aplikace implementuje vizualizační knihovnu v podobě webview (webové zobrazení).

Na obrázku 4.1 je viditelná závislost mezi všemi komponentami. Rovněž je zde patrná základní struktura těchto knihoven a jejich závislosti mezi sebou. Avšak jsou zakresleny jen ty komponenty, které je možné považovat za důležité. Je dobré zdůraznit asynchronní komunikaci, kterou poskytuje knihovna vyhodnocující regexy. Tuto komunikaci lze vidět mezi komponentou Regexer a RegexVisualizer s tím, že Regexer využívá vedlejší vlákno mimo vlákno hlavní. Knihovna pro vlákna threads.js je blíže popsána v sekci 4.3. Rozšíření VSCode v rámci webview, zobrazuje HTML soubor index.html z vizualizační knihovny. Také rozšíření může komunikovat s touto knihovnou pomocí zpráv, pro předávání dat a pro vyvolávání událostí. RegexMatch je pak využíván vizualizační knihovnou, pro zobrazování stavů vyhledávání.



Obrázek 4.1: Struktura knihoven aplikace

4.3 Použité technologie

Celá aplikace je integrovaná do vývojového prostředí **Visual Studio Code**. Jádru aplikace je psáno v programovacím jazyce **TypeScript** (*TS*) verze 5.3, který rozšiřuje jazyk **JavaScript**, zkráceně *JS*. TypeScript má jako hlavní nástavbu možnost využívání a přiřazování datových typů. Také platí, že kód napsaný v JS je správný v rámci TS, ale to neplatí naopak. Psaní větší aplikace je často vhodnější v TS, kvůli svým typovým kontrolám, čímž se jako programátor mohu vyvarovat potencionálním chybám při běhu programu. Také vývojové prostředí VSCode, zpřístupňuje API pouze pro JavaScript nebo TypeScript. Sice by bylo možné mít část aplikace napsané v jiném jazyce, to by ale mělo své komplikace při vývoji.

Pro parsování jsem se rozhodl použít bezkontextovou gramatiku **Peggy**[10, 11], pro jazyk JavaScript. Ta umožňuje poměrně snadného zpracování textové podoby regulárních výrazů, do podoby nějaké struktury. Tato výsledná struktura může být v podstatě jakákoliv.

Všechny části aplikace jsou spravovány balíčkovým manažerem **NPM** (Node Package Manager). Také tyto části využívají některých balíčků, které jsou dostupné pro npm. Určitá část aplikace je postavená na technologii **NodeJS**. Jedná se o JavaScript runtime (běhové prostředí), typicky určené pro serverové aplikace. Například runtime VSCode rozšíření je NodeJS, oproti tomu samotné webview běží ve webovém runtime, které je typické pro webové prohlížeče.

Vizualizační část aplikace pak využívá základní **HTML** (HyperText Markup Language) struktury. HTML je základem pro webové stránky a definuje jejich strukturu pomocí značek. Pro následnou změnu vzhledu (stylu) jsem využil technologie **LESS**⁸, což je rozšíření standardního **CSS** (Cascading Style Sheets). Avšak LESS musí být transpilovaný⁹ do CSS, jelikož webové prohlížeče ho neumí zpracovat. LESS umožňuje například vnořování stylů nebo tvorbu vlastních proměnných. Pro logickou část vizualizační knihovny je také využit TypeScript.

Pro výsledný přeložený kód je použit balící nástroj **Webpack**¹⁰. Ten mi umožňuje všechny části aplikace poměrně efektivně zabalit do malého počtu souborů. Tento nástroj se pak hodí pro menší výslednou aplikaci a hlavně pro seskupení všech závislostí. Mohu mít i větší kontrolu nad výsledným kódem. Například lze udávat, kdy se mají soubory dělit, jak se mají zpracovávat přílohy, jako jsou obrázky atd. Pro optimalizaci a úpravu kódu se zde využívá takzvaných *loaderů* a *pluginů*, které dokážou v určité části překladače zasáhnout a popřípadě změnit určitou část kódu. Ve výsledku se jedná o velice silný nástroj, který dává programátorovi větší kontrolu nad výsledným přeloženým kódem aplikace.

Jelikož jsem chtěl mít větší jistotu, co se týče správnosti aplikace, je v algoritmické části aplikace využito technologie pro tvorbu testů. Tato knihovna se nazývá **Jest**¹¹. Avšak tato knihovna

⁸<https://lesscss.org/>

⁹Typ překladače z jednoho jazyka na jazyk jiný.

¹⁰<https://webpack.js.org/>

¹¹<https://jestjs.io/>

slouží převážně pro testování JavaScriptových kódů, proto jsem k ní využil balíčku **TS-Jest**¹², pro TypeScript. To mi pak umožňuje psát testy, které mohou využívat TypeScriptové typy.

Jednou z posledních knihoven, kterou jsem použil, je **Threads.js**¹³. Jelikož existují různé runtime JavaScriptu, tak neexistuje jednotný standard pro implementaci vláken (threads). Browser má tzv. *web workery* a NodeJS má *worker threads*, sice si jsou podobné, ale mají změny které znemožňují univerzálního použití. Proto je v této aplikaci využito threads.js, které eliminuje tyto problémy. Navíc dokáže nabídnout větší bezpečnost pro programátora, který píše kód v TS. Tato bezpečnost je docílená tím, že knihovna dokáže poskytnout z vlákna rozhraní, které může obsahovat také typy.

¹²<https://www.npmjs.com/package/ts-jest>

¹³<https://threads.js.org/>

Kapitola 5

Knihovna pro práci s regulárními výrazy

Před tvorbou samotné vizualizace jsem měl úvahu o použití knihovny, která by mi umožňovala získat zpracovanou strukturu historie průchodu regulárním výrazem. Sice implementace regulárních výrazů se nachází v samotné specifikaci JavaScriptu, ale ta mi neumožňuje získat informaci o samotném vyhledávání. Po prozkoumání existujících řešení, která by vyhovovala této práci, jsem se rozhodl vytvořit vlastní implementaci v podobě této knihovny. Nenalezl jsem totiž řešení, které by bylo dostatečně flexibilní a zároveň lehce integrovatelné do programovacího jazyka TypeScript. Sice vlastní implementace může být pracná, ale jelikož chci mít co nejvyšší kontrolu nad výslednou strukturou, tak jsem toto řešení vyhodnotil za vhodné.

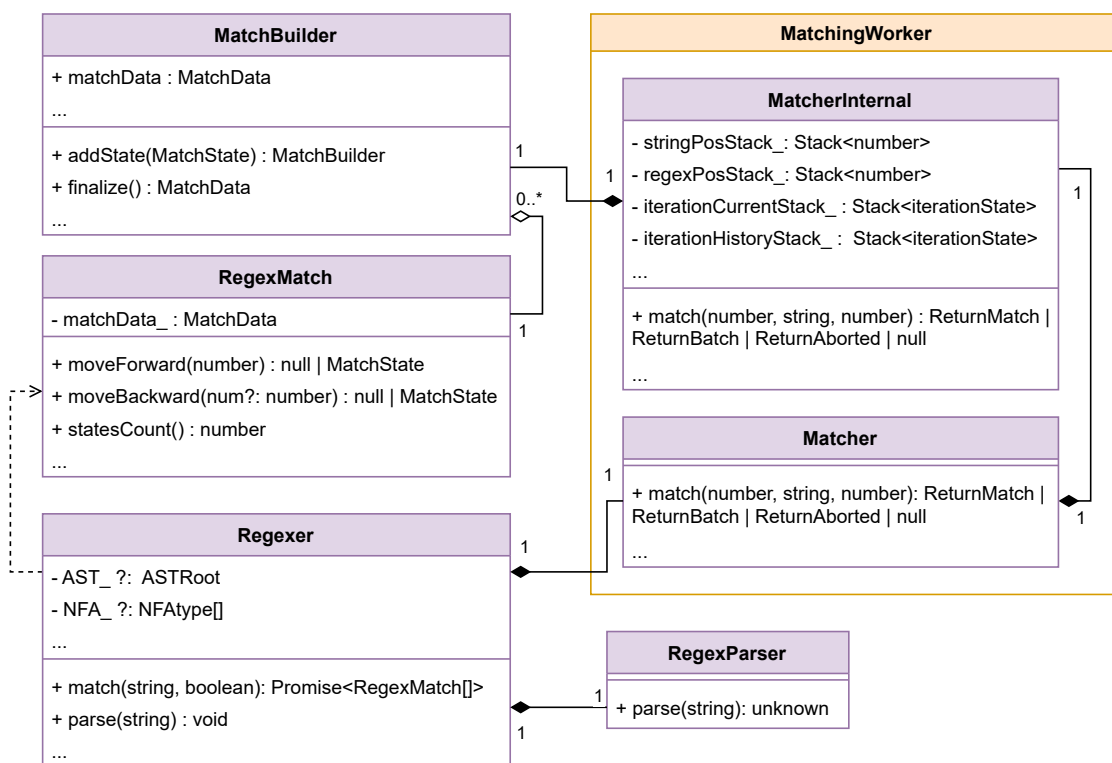
Nejprve v podkapitole 5.1 se pokusím vysvětlit návrh této knihovny. Dále princip parsování regulárních výrazů v podkapitole 5.2. Nakonec v podkapitole 5.3 popíšu funkčnost vyhledávání s jeho výsledkem.

5.1 Architektura

Vstupní třídou pro tuto knihovnu je **Regexer**. Propojuje jednotlivé části této knihovny a také poskytuje její rozhraní. Dále si drží důležité informace, které souvisí s aktuálně zpracovaným regulárním výrazem. *RegexMatch* je třída, která reprezentuje jeden výsledek vyhledávání zadaným výrazem. Její data jsou soukromá, ale umožňuje je procházet pomocí svých metod. Data této třídy jsou generována třídou *MatchBuilder*. Její instance existuje pouze ve chvíli, kdy probíhá vyhledávání v zadaném řetězci. Rozhraní této třídy umožňuje přidávání stavů, které mají v rámci vyhledávání význam informace o aktuálním stavu v regulárním výrazu a hledaném řetězci. Dále třída *MatcherInternal* má za úkol průchod zadaným řetězcem pro konkrétní výraz. Tato třída je izolována a není dostupná z vnější, jak její název *internal* (česky vnitřní) napovídá. Obsahuje hlavní algoritmickou část průchodu nedeterministickým automatem. Naopak třídou, která poskytuje viditelné rozhraní a volá metody třídy *MatcherInternal*, je *Matcher*. Její rozhraní je poskytováno třídě *Regexer*. Pro parsování textové reprezentace regulárního výrazu na strukturu NKA a AST, slouží rozhraní *Re-*

gexParser. Jedná se o parser, který vzniká vždy po překladu bezkontextové gramatiky. Stack je jednoduchá struktura, která je ale velmi důležitou součástí vyhledávání. Zásobník programu umožňuje zbavit se rekurzivního volání funkce. Rekurse obecně vede k pomalejšímu chodu programu a nelze ji jednoduše pozastavit v jakémkoliv čase a stavu. Také může jednoduše při složitějším zpracování dojít k přetečení zásobníku, který je často limitován, aby nedošlo k nekonečnému rekurzivnímu volání. Sice rozhraní pole v JS je připraveno na funkcionalitu zásobníku, ale nezaručuje programátorovi striktní pravidla pro zásobník. Z tohoto důvodu jsem zvolil jednoduchou implementaci zásobníku, která omezuje manipulaci se základním polem, na operace určené pro zásobník.

Vztahy mezi jednotlivými třídami lze vidět na obrázku 5.1. Nachází se zde také obalující blok **MatchingWorker**, který reprezentuje vstupní soubor vedlejšího vlákna, jež slouží pro asynchronní komunikaci s hlavním vláknem. *Regexer* jako vstupní třída uchovává informace o AST a NKA regulárního výrazu. Poskytuje také dvě hlavní metody, a to *parse* pro zpracování výrazu a *match* pro vyhledání v textu. Po zavolání metody *match* se zavolá metoda třídy *Matcher* pro zpracování výrazu.



Obrázek 5.1: Třídní diagram části knihovny pro práci s regulárními výrazy

5.2 Parsování regulárních výrazů

Parser

Jak již bylo zmíněno, pro parsování regulárních výrazů jsem použil bezkontextovou gramatiku *Peggy*. Jedná se o pokračování projektu PegJS, ale ten se již dlouho nevyvíjí. Jelikož knihovna Peggy je stále aktualizována a má velkou podporu vývojářů, tak jsem zvolil její využití pro tuto práci.

V ukázce 5.1 se nachází vstupní neterminál bezkontextové gramatiky *start*. Ten obsahuje výběr mezi dvěma začátky *moded_start* a *general_start*. Výběr je pak dostupný pod názvem *type*, podle toho který se zvolí. Před dokončením pravidla se na jeho konci může nacházet blok "{", který může modifikovat výsledná data. Modifikace v rámci pravidla pro začátek probíhá zavoláním instance vlastní třídy **ParserHandler**, která je součástí gramatiky. Třída má za úkol zpracovávat příchozí data do struktury, která je ukázaná na obrázku 5.2.

```
start
=
type: (moded_start / general_start)
{
    const data = { modifiers: type?.modifiers };
    return handler.handle(data, type?.elements, States.ROOT);
}
```

Zdrojový kód 5.1: Jednoduché pravidlo gramatiky

Existuje několik různých vzorů regulárních výrazů. Každý ze vzorů má pravidla pro možné kombinování s ostatními vzory. Ukázka kódu 5.2 obsahuje příklad možných výběrů pravidel, které lze společně kombinovat. Například možnosti pro iteraci, ve zmíněném kódu *to_iterate*, obsahují pouze následující vzory, které mohou být opakovány.

- Speciální znaky (*escaped_special*) — "\s", "\d"
- Základní znaky (*primitive*) — "a", "b", "0"
- Výběr jakéhokoliv znaku (*any_character*) — "."
- Skupina (*group*) — "("
- List znaků (*list*) — "[a - z]"

V mnoha případech záleží na pořadí výběru z dostupných vzorů, proto je potřeba určit, které možnosti upřednostnit. Abych vysvětlil, proč je pořadí důležité, vybral jsem si jako příklad **iteraci**

(iteration) a **výběr** (option). Zjednodušeně výběr má vyšší přednost, jelikož může mít za potomka iteraci. Kdyby se neterminál iterace nacházel před neterminálem výběru, tak by došlo k tomu, že by iterace nebyla součástí výběru, v případě, že by se nacházela na pozici první možnosti výběru. Neboli byla by dříve zpracována, nežli samotný výběr. Jako příklad může sloužit výraz $a * |b*$, při kterém by se první zpracovala iterace $a*$. Výsledkem výběru by byly dvě možnosti ϵ nebo $b*$, což je sice sám o sobě správný tvar výběru, ale ve zvoleném výrazu **musí** být výsledný výběr $a*$ nebo $b*$.

```
any_element
    = option / iteration / optional / general

to_iterate
    = escaped_special / primitive / any_character / group / list
```

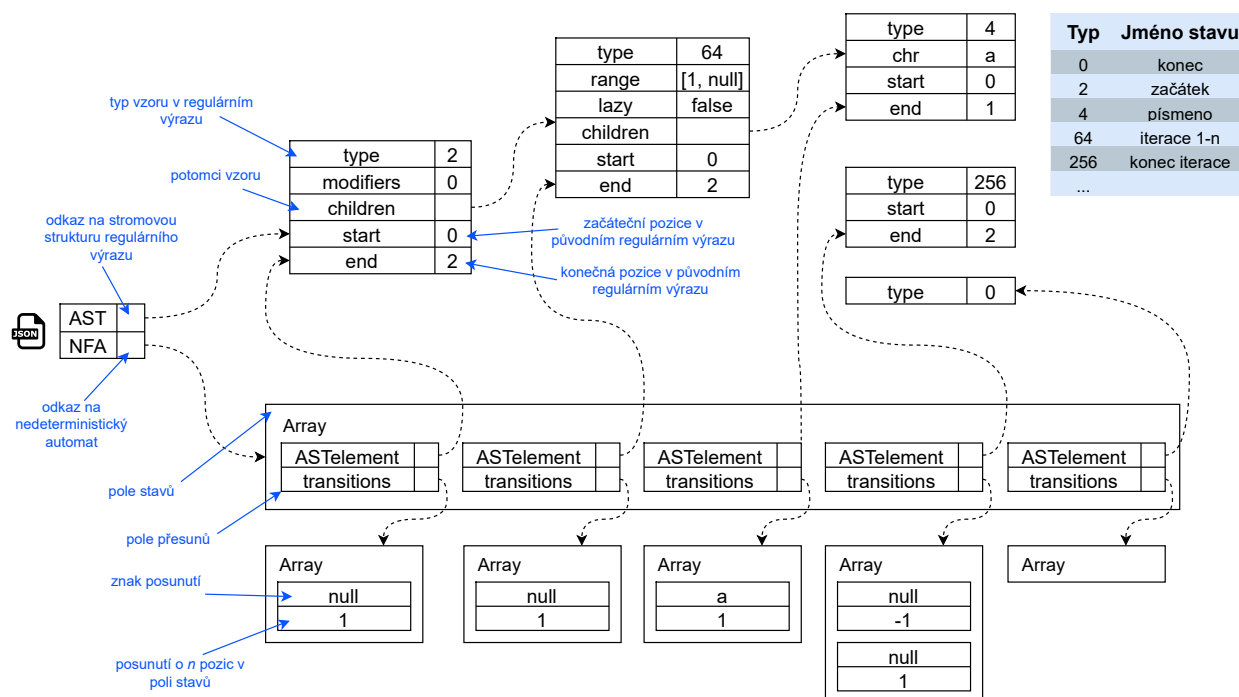
Zdrojový kód 5.2: Výběry neterminálů pro některé vzory regulárních výrazů

Struktura zpracovaného regulárního výrazu

Pro zpracovaný regulární výraz jsem zvolil strukturu, která obsahuje **nedeterministický konečný automat** (NKA), zároveň s **abstraktním syntaktickým stromem** (AST). AST pak slouží k dohledání informací o původním regulárním výrazu. Výsledná struktura je objektem, který obsahuje pouze data a má výchozí prototype¹⁴. Tuto strukturu je možné vidět na obrázku 5.2. NKA je ve formě **přesunové tabulky (transition table)**. Ta má tvar pole, kde každá položka obsahuje informaci o konkrétním stavu a přesunech na další stavy. Stav je identifikovaný na základě indexu v poli. Přesuny jsou pak implementovány tak, že každý stav si uchovává všechny své přesuny, které vedou z daného stavu do stavu jiného. Každý přesun pak má informaci, o jaký znak přesunu se jedná a na jaký index (stav) v poli odkazuje.

Na obrázku 5.2 lze vidět výslednou strukturu pro výraz $a+$, která obsahuje dva atributy AST a NFA. Klíč NFA odkazuje na pole stavů přesunové tabulky. AST má odkaz na počátek struktury, který signalizuje začátek regulárního výrazu. Je zde patrné, že každý stav v tabulce přesunů má odkaz na odpovídající prvek v AST. AST prvek/vzor drží různé informace, např. pozice v původním řetězci (start a end), potomky daného stavu nebo typ vzoru. Ne každý stav musí mít potomky, ale například skupina potomky má. Některé vzory, jako je například iterace, obsahují v AST dodatečné informace, jako je rozsah iterace (range). Typy stavů jsou číselné hodnoty, a ty které se nacházejí na obrázku jsou zde také popsány.

¹⁴https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes



Obrázek 5.2: Příklad výsledné struktury regulárního výrazu a+

5.3 Vyhledání pomocí regulárního výrazu

Vyhledání je jednou z hlavních částí této knihovny, jedná se o procházení regulárním výrazem a hledaným řetězcem. Výsledkem je struktura dat, která obsahuje informace o zpracovaném vyhledávání. V této části textu popisují, jak jsem naimplementoval vyhledávání, důležité koncepty a výslednou strukturu.

Odstranění rekurze

Rekurze je sice důležitým aspektem mnoha programů a dokáže usnadnit některé problémy, ale existují situace kdy se vyplatí jí zbavit. V první řadě, bych rád vysvětlil, proč je rekurzivní řešení vhodné pro vyhodnocování regulárních výrazů. Jak jsem již zmínil, tak v regulárním výrazu může dojít k backtrackingu. Nastane ve chvíli, kdy není možné z daného stavu v NKA, přejít na stav jiný. V tuto chvíli, dojde k vrácení se v NKA do předchozího stavu a k pokračování vyhledávání pomocí další možné cesty. Nejjednodušší řešení tohoto případu, je použití rekurze. Pro představu, přechod značí rekurzivní volání funkce a pokud není možné přejít do dalšího stavu, tak se vrací do předchozího volání funkce.

Rekurzi lze odstranit pomocí zásobníků, nebo-li program si uloží, jen potřebné informace. Ve chvíli kdy dojde k vrácení se (backtrackingu), tak se odstraní vrch zásobníků. Důležité tedy je správně řídit správu zásobníků, což může být lehce komplikované.

Úryvek zdrojového kódu 5.3, obsahuje základní vkládání nově navštíveného stavu do zásobníku. Přesněji se zde ukládá jak stav, tak také index 0 značící počáteční přesun. Pokud se stav již nachází na vrchu zásobníku, tak je pouze navýšen index přesunu.

```
const nfaState = NFA[<number>this.regexPosStack_.top()] as NFATYPE;
let topState = this.statesStack_.top();
if(topState?.state !== nfaState)
    this.statesStack_.push({transition: 0, state: nfaState});
else
    topState.transition++;
```

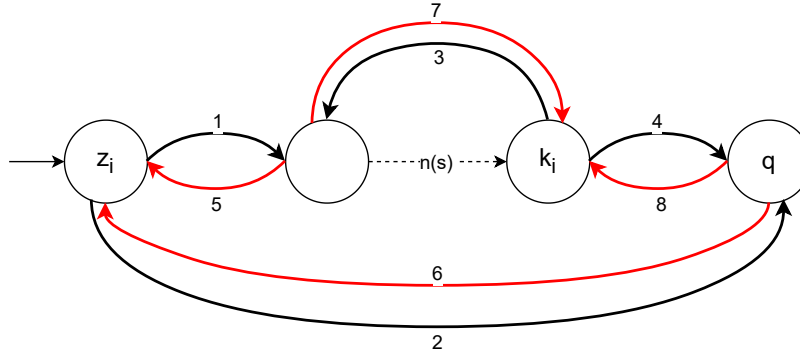
Zdrojový kód 5.3: Vložení stavu do zásobníku

Příklad kódu 5.4, souvisí s předchozí ukázkou. Jestli nastane situace, kdy neexistuje žádný další přesun z aktuálního stavu, je vyvolán backtracking. Metoda *handleBacktracking* se stará o správu backtrackingu. Převážně se jedná o odebírání vrchů zásobníků, jako je již zmíněný zásobník stavů. Pokud není vrácená hodnota **null**, znamená to ukončení nebo pozastavení vyhledání. K neúspěšnému vyhledání a následnému ukončení dojde, pokud je zásobník stavů vyprázdněný a pozice v hledaném řetězci se nachází na jeho konci.

```
const transitions = (nfaState as NFATYPE).transitions;
if(transitions.length <= <number>this.statesStack_.top()?.transition)
{
    const returned = this.handleBacktracking();
    if(returned !== null) return returned;
    continue;
}
```

Zdrojový kód 5.4: Vyvolání backtrackingu, pokud neexistují další přechody ze současného stavu

Počítání iterací a prevence nekonečných cyklů



Obrázek 5.3: NKA pro popis počítání iterací a prevenci nekonečných cyklů

V seznamu pravidel, I_i značí identifikátor iterace, i je počet dokončených opakování iterace a P_s je aktuální pozice v hledaném řetězci. I_{arr} obsahuje informace v poli o jedné iteraci $[I_i, i, P_s]$.

Algoritmus pracuje se dvěma zásobníky, určenými pro držení informací o iteracích. První uchovává aktuálně nedokončené, resp. probíhající iterace. Pro popis jej označuji Z_s (zásobník současných iterací). Druhý značím jako Z_h (zásobník historie), ten slouží pro iterace, které byly již dokončené. Historie je důležitá pro backtracking.

- 1 – Vlož do $Z_s \leftarrow [I_i, 0, P_s]$
- 2 – Vlož do $Z_h \leftarrow [I_i, 0, P_s]$
- 3 – Vrchol $Z_s, I_{arr}[1] + 1$
- 4 – Vlož do $Z_h \leftarrow$ Odeber ze $Z_s, I_{arr}[1] + 1$
- 5 – Odeber ze Z_s
- 6 – Odeber ze Z_h
- 7 – Vrchol $Z_s, I_{arr}[1] - 1$
- 8 – Vlož do $Z_s \leftarrow$ Odeber ze $Z_h, I_{arr}[1] - 1$

Na obrázku 5.3 lze vidět nedeterministický automat. Jedná se o obecnou reprezentaci iterace, kde z_i reprezentuje začátek iterace, k_i konec iterace a q značí první stav za iterací. Mezi z_i a k_i se nachází množina stavů $n(s)$. Červené šipky signalizují backtracking a černé značí klasický přechod mezi stavy.

Jelikož existují iterace v rozmezí, například od 3 do 6, tak je potřeba znát informaci, v kolikátém opakování se právě konkrétní iterace nachází. Pro tento problém jsem zvolil 8 pravidel, které popisují

řešení osmi různých přesunů mezi stavy. Tato pravidla jsou rozepsána pod obrázkem 5.3, indexy pravidel korespondují s indexy v obrázku. Rád bych poukázal na to, že při backtrackingu se vždy zásobníky vrací, do původních stavů. To znamená, mám-li například stavy a a b , tak platí pro přesun $a \rightarrow b$ a pro následující backtracking $b \rightarrow a$, že ve stavu a musí být po dokončení obou přechodů hodnoty zásobníků ve stejném stavu jako na začátku. Jedno opakování je dokončeno při přechodu 3 nebo 4. Zároveň přechod 4 společně s přechodem 2, jsou konečnými přechody pro danou iteraci.

Zásobník současných iterací obsahuje poměrně malé množství informací, jelikož se jedná pouze o probíhající iterace. Naopak zásobník historie může obsahovat poměrně hodně informací. Má-li iterace například 100 dokončených opakování, tak historie bude obsahovat minimálně 100 záznamů. To se může zdát jako mnoho zbytečných informací, ale nelze předem prakticky vědět jestli dojde k backtrackingu a kde se zastaví.

Další důležitou kontrolou, kterou je nutné splnit, je na konci iterace zkontrolovat zda se nachází v určeném rozmezí. Jelikož počítám jejich opakování, tak stačí tuto informaci porovnat s náležitými mezemi.

V některých případech by mohlo dojít k nekonečnému cyklu. Například pro regulární výraz $()+$ by k tomu došlo tak, že by nenastalo k posunu v hledaném řetězci, ale přes to by stále procházela úspěšně dále. K tomu slouží ukládání poslední pozice v hledaném řetězci, při začátku nové iterace, nebo zopakování. Jestli má dojít k zopakování, musí proběhnou kontrola, zda-li došlo ke změně pozice v řetězci od posledního zopakování. V obrázku 5.3 se jedná o stav k_i a přesun 3.

Využití vlákna pro vyhledávání

Nedílnou součástí této knihovny je **paralelní zpracování** v podobě balíčku `threads.js`. Balíček byl již zmíněn v kapitole 4.3. *Pralelismus* dovoluje složité operace přesunout do vedlejšího vlákna, aby hlavní vlákno nebylo zatěžováno. Vlákna sice umožňují efektivnější zpracování náročných programů, ale také mají svá úskalí.

S volbou vývojového prostředí *VSCode*, byla nutnost splnit podmínky stanovené pro práci s web workery, v souladu s jejich API [12]. Podmínkou totiž je, nutnost mít zdrojový kód workeru přímo vložený ve zdrojovém kódu hlavního vlákna. To znamená, že worker nesmí být přímo načítaný, z adresáře rozšíření. Avšak tato nutnost, je komplikovaná a proto následovně vysvětlím, jak jsem tento problém řešil.

Všechny závislosti, které worker má, musí být součástí jednoho výsledného souboru. To je docíleno tím, že přeložím soubor pomocí *webpack*, který vytvoří jeden výsledný soubor. Pokud by někdo chtěl využít této knihovny, v rámci prostředí NodeJS nebo Prohlížeče, tak tento překlad probíhá dvakrát pro oba runtime. Tento soubor může následně být vložen přímo do zdrojového kódu. Pokud aplikace, která využívá tuto knihovnu má *webpack*, může využít loaderu, který jsem pro tuto knihovnu napsal. Ten dokáže v místě kde je worker volaný, vložit jeho zdrojový kód, v rámci textového řetězce. Výsledkem je worker, který je vložený jako řetězec, ve zdrojovém kódu hlavního vlákna.

Původně tato knihovna zprostředkovávala, pouze paralelismus pro prostředí *NodeJS*. Později se ale ukázalo, že tato restrikce je limitující, co se týče využití této knihovny ve vizualizační části aplikace. Regulární výrazy se totiž zpracovávali, na straně rozšíření, které běží v prostředí *NodeJS*. Vizualizace byla tedy omezena na komunikaci s prostředím, co se týče práce s regulárními výrazy. Vývojové prostředí pak sloužilo, jako komunikační uzel mezi vizualizací a touto komponentou. Také toto zprostředkování, poměrně zpomalovalo výkon aplikace, jelikož se musela data posílat pomocí zpráv mezi dvěma komponentami. Samotná vizualizace, nemohla existovat jako samostatná webová stránka, jelikož zde existovala přímá závislost na rozšíření. Změnou této části aplikace, která současně podporuje využití paralelismu, pro jakákoliv prostředí, byly tyto problémy eliminovány.

Výsledek vyhledávání

Výsledkem vyhledávání je třída, obsahující data s informacemi o procházení. Jejich tvar se neřídí žádným standardem, nebo-li výsledná struktura je čistě přizpůsobená této práci. Vlastnosti výsledného objektu obsahují všechny důležité informace. První hodnotou je, zda-li bylo vyhledání úspěšné, či nikoliv. Další jsou skupiny, které drží informace, kde se nachází v regulární výrazu a hledaném řetězci. Pokud se jedná o pojmenovanou skupinu, tak se také ukládá její jméno. Poslední vlastností, která stojí za zmínku je pole, nebo-li seznam všech po sobě jdoucích stavů.

Ve stavech se nachází údaje, které reprezentují historii průchodu. Každý stav obsahuje, údaj o pozici v řetězci a ve výrazu. Také musí být identifikován, o jaký vzor regulárního výrazu se jedná. Stav může obsahovat další data, která jsou nepovinná, nebo-li se nenachází ve všech stavech. Jedná se převážně o typ akce a seznam skupin. Akce je informace, která upřesňuje typ stavu, jako je například backtracking. Seznam skupin se může nacházet, také v jednotlivých stavech. Lze pak pozorovat průběh vývoje skupin, s vývojem stavů.

Výsledné stavy se mohou lišit, jak dle počtu, nebo také podle tvaru. Modifikace vznikne na základě předem určených nastavení. Ta například umožňují zahodit nežádoucí informace, nebo naopak přidat rozšiřující. Zvolil jsem tuto možnost nastavení, aby knihovna mohla být univerzálnější a flexibilnější.

Data jsou uložena v objektu, který je dále součástí třídy **RegexMatch**. Samotná třída poskytuje pouze rozhraní pro procházení stavů, nebo popřípadě pro získání dalších základních informací o vyhledávání.

Kapitola 6

Vizualizační knihovna a rozšíření

Tato část aplikace se zabývá, knihovnou určenou pro vizualizaci průchodů regulárními výrazy a rozšířením do vývojového prostředí *VSCo*de. Pro získání informací o vyhledávání, slouží již zmíněná knihovna, která byla popsána v předchozí kapitole 5. Hlavním cílem této části aplikace je, na implementovat uživatelsky přívětivé a intuitivní rozhraní, integrované do vývojového prostředí. To umožňuje zadávat regulární výrazy, text ve kterém lze pomocí zadaného výrazu vyhledávat a následnou vizualizaci ve formě debuggeru. Vizualizace je koncipována, jako webová stránka, která je zobrazená do prostředí pomocí tzv. *webview*, ale také může existovat samostatně.

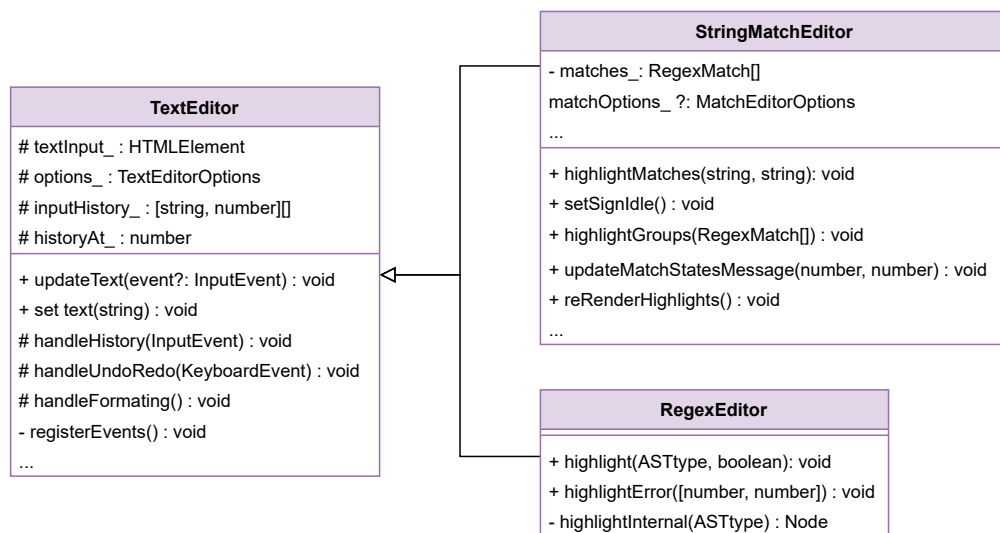
6.1 Návrh vizualizační části

Vstupním souborem knihovny je HTML soubor *main.html*. Ten také vkládá skript *index.ts*, jehož hlavním účelem je inicializovat vše potřebné pro chod aplikace. Hlavní třídou, která se stará o obsluhu vizualizace je **RegexVisualizer**. Jejím úkolem je obsluhovat ostatní komponenty a přímo komunikuje s třídou *Regexer*, ze které následně získává data pro vizualizaci. Pro interakci uživatele slouží dva textové editory, ty jsou ve formě dvou tříd *RegexEditor* a *StringMatchEditor*. Obsluhují HTML elementy pro zadávání textu, jejich základní funkcionalita je děděna ze třídy *TextEditor*. Pro samotnou vizualizaci ve formě ladícího nástroje, existuje třída *RegexDebugger*. Má za úkol, obsluhovat okno aplikace, kde se samotný debugger nachází. Obsahuje i vlastní posuvník (*Slider*), který dokáže vyvolávat události, obsahující informace o aktuální hodnotě posuvníku. Také například disponuje možností automatického přehrávání a změnou rychlosti.

6.2 Implementace textových editorů

Ve vizualizaci se nachází dva textové editory. První slouží pro zadávání regulárního výrazu a druhý pro zadávání řetězce, ve kterém proběhne vyhledávání, na základě zadaného výrazu. Jejich funk-

cionalita je obalena ve dvou třídách, každá sloužící pro jiný textový editor a obě dědí ze třídy *TextEditor*. Tento vztah lze vidět na obrázku 6.1.



Obrázek 6.1: Třídní diagram textových editorů

Základní řešení

Pro řešení textových editorů, jsem se rozhodl pro vlastní implementaci, pro větší flexibilitu a řešení konkrétních problémů týkajících se práce s regulárními výrazy. Textový editor umožňuje rozšířené možnosti práce s textem oproti HTML elementům, jako jsou input nebo textarea. Tyto možnosti jsou například vlastní formátování, nebo správa historie. K realizaci samotných vstupů, jsem použil základní obalující HTML blok `span`. Na zvoleném bloku tolik nezáleží, ale je potřeba, aby měl atribut `contenteditable` s nastavenou hodnotou na `true`. Tento atribut povoluje psaní přímo do daného bloku. Oproti elementu jako je input, lze zde vkládat HTML kód a tím upravovat formátování textu. To je vhodné pro regulární výrazy, jelikož sami o sobě nejsou moc přehledné.

TextEditor slouží jako vzorová třída, pro realizaci textových editorů. Drží si referenci na HTML element, který obsluhuje, pod názvem `textInput_`. Pro lepší interaktivitu s tímto elementem, je potřeba zaregistrovat různé události. Mezi ně patří např. psaní, mazání, undo a redo. Události jsou registrovány při vytvoření instance třídy, pomocí soukromé metody `registerEvents`. Pokud je zavolána, chráněná metoda `handleFormatting`, tak dojde ke změně podoby textu na formátovanou. Jedná se o grafické zobrazení bílých znaků, jako je nový řádek nebo tabulátor.

Komunikace s třídou *Regexer*

Komunikace funguje pomocí vyvolání událostí z textových editorů. Je-li událost vyvolána editorem pro regulární výraz, tak se pomocí třídy *Regexer* zpracuje daný výraz. Pokud dojde k vyvolání

události změny textu v jednom ze dvou editorů, tak dojde k zavolání metody pro vyhledávání (match).

Vyhledávání pak funguje na bázi asynchronní komunikace, kdy se výsledky posílají po tzv. dávkách (anglicky batch). Výhodou této komunikace je to, že pokud uživatel v průběhu zpracovávání změnil text jednoho z editorů, tak se proces ukončí a tím pádem není potřeba čekat na jeho dokončení. Velikost jedné dávky jsem zvolil na 20000 stavů, jelikož tento počet stavů, je poměrně rychle vyhodnocený. Informace o zpracování, se aktualizují průběžně po každé příchozí dávce.

Zvýraznění syntaxe

Součástí třídy *RegexEditor*, je metoda sloužící pro zvýrazňování syntaxe regulárních výrazů. Pro zvýraznění slouží získaná AST struktura po dokončeném parsování. Algoritmus řešení tohoto problému, funguje na principu rekurzivního zanoření, ve stromové struktuře. Každý symbol, který má být zvýrazněný je obalený v HTML bloku, s třídou identifikující o jaký symbol se jedná. U některých vzorů, záleží na pořadí zpracování symbolů a rekurzivního zanoření. Například skupina se zpracovává tak, že první se zvýrazní otevírací závorka "(". Poté se algoritmus rekurzivně zanoří, nebo-li zpracuje potomky (vnitřní část) skupiny a nakonec zvýrazní ukončující závorku ")". Výsledkem vznikne HTML struktura, která popisuje symboly a vzory regulárních výrazů. Tyto symboly jsou pak zvýrazněny pomocí různých barev definovaných v CSS.

Historie

Implementace historie textových editorů je vlastní, jelikož původní nefungovala správně. Důvodem bylo časté přepisování textu, z již zmíněného formátování. Aby historie správně fungovala, tak jsem musel vytvořit pole, které obsahuje jak původní řetězec tak pozici kurzoru v něm. Pokud je vyvolána operace vrácení se zpět v historii (undo), přkopíruje se uložený řetězec do textového pole a kurzor se nastaví na správnou pozici. K odstranění záznamu z historie nedochází, jelikož může být vyvolána operace *redo*, nebo-li odvolání operace *undo*. Pokud dojde k uložení nového stavu textového pole, tak všechny stavy za ukazatelem se smažou a přidá se zde nový. V nastavení editoru, jsem přidal možnost zvolit si maximální počet záznamů v historii. Pokud ale není nastavená, automaticky se omezí na 100 záznamů.

Pozice textového kurzoru

Práce s textovým kurzorem je další značná část textových editorů. Pokud uživatel píše do textového pole, tak často nastává změna textu na pozadí samotnou aplikací. Například při zvýraznění, dochází ke změně textové formy na HTML strukturu. Při změně vždy dojde k resetování pozice kurzoru v textu. To ale pro uživatele není příjemná vlastnost, kterou jsem tedy musel vyřešit.

Před přepsáním textového pole, je uložena pozice kurzoru. Po vložení textu, je nutné vrátit se na uloženou pozici. Nejedná se ale o jednoduchou úlohu, jelikož pokud se v textu nachází HTML

elementy, musí být brány v potaz. Použil jsem základ algoritmu ze stránky *stackoverflow*¹⁵, který dokáže jak zjistit aktuální pozici kurzoru, tak z pozice umístit kurzor na správné místo. Ten jsem upravil pro potřeby mého projektu a dále rozšířil. Například jsem přidal možnost vytvoření nového kurzoru, který není přímo vložený do dokumentu, což se například může hodit pokud je potřeba získat souřadnice písmena.

6.3 Vizualizace průchodu

Vizualizace ve formě debuggeru je obsluhována třídou *RegexDebugger*. Okno pro vizualizaci se otevře po kliknutí na tlačítko, jehož reference je předána třídě součástí konstruktoru. Debugger obsahuje identická pole jako textová pole pro interakci s uživatelem, avšak již nelze jejich text editovat. Dále disponuje posuvníkem, který slouží pro procházení průběhu vyhledávání.

Posuvník

Posuvník jsem zvolil, jako jednoduchou a intuitivní možnost procházení historie. Jeho implementace je ve vlastní třídě a jeho součástí je nastavení, pro příjemnější změnu výsledného posuvníku. Pomocí nastavení, lze vypínat/zapínat některé funkcionality, nebo měnit samotný vzhled, jako je barva či velikost. Tato realizace je vlastní, z důvodu lehčí integrace do aplikace.

Při vytváření instance této třídy, musí být předán HTML element nebo id elementu, do kterého se posuvník vykreslí. Nastavení je dobrovolné, pokud není předáno zvolí se základní. Posuvník může obsahovat tlačítka pro ovládání, kterými jsou automatické přehrávání, pozice vpřed, pozice zpět, konec a začátek posuvníku. Pro automatické přehrávání, může být součástí posuvníku pole pro editaci rychlosti, pokud je povolené v nastavení.

Posuvník může nabývat pouze celočíselných hodnot, v omezeném rozmezí od *min* do *max*. Pokud se změní jeho hodnota, tak je vyvolána vlastní událost, která tuto hodnotu obsahuje. Ta může být odchycena např. jinou třídou.

Zvýraznění pozice a backtrackingu

Pro vizualizaci slouží zvýraznění pozice, jak v regulárním výrazu, tak v hledaném řetězci. Pozice je zvýrazněná tak, že se v pozadí pozice nachází barevný blok, který je vykreslený do HTML plátna (canvas). Řešení tohoto problému jsem několikrát změnil, jelikož se původní řešení ukázalo neúčinným v některých případech.

Jako první řešení, jsem zvolil získání šířky písmene, výšky řádku a velikost mezery mezi písmeny. Poté jsem procházel celý řetězec a pokud byl znak součástí pozice pro zvýraznění, tak jsem rozšířil šířku zvýrazněného bloku o šířku písmene a velikost mezery. Jestli byl nalezen znak nového řádku, nebo délka zvýrazněného bloku přetekla velikost řádku, tak jsem vytvořil nový blok pro nový řádek.

¹⁵<https://stackoverflow.com/questions/69956977>

Problémem tohoto řešení bylo to, že když došlo k nekontrolovanému zalomení řádku tzn. pokud řádek byl dříve zalomen než konec tohoto řádku. To mohlo například nastat při zalomení slova na nový řádek. Ve výsledku docházelo k zvýraznění prázdného místa a také k jeho nesprávnému konci.

Druhé řešení, které jsem zkusil na implementovat, bylo pomocí využití textového kurzoru. Princip byl již jiný, jelikož nebylo třeba znát velikost písmene a mezery. Kurzor jsem nejprve umístil, na začáteční pozici zvýraznění. V cyklu, jsem postupně posouval kurzorem až na konec zvýraznění. Během tohoto procesu jsem si ukládal souřadnice, kde se nachází. Tento způsob již zamezil problému při zalamování řádku, ale byl poměrně neefektivní a dokázal zpomalovat uživatelské rozhraní.

Poslední způsob implementace, dokázal vyřešit i zmíněný problém se zpomalením. Vychází z předchozího popisu, jelikož také využívá kurzoru. Rozdílem je, že kurzor je vložen jako rozsah od začátku až po konec zvýraznění. Není tedy třeba procházet, písmeno po písmenu. Kvůli toho jsem upravil kód, pro získání a nastavení pozice kurzoru, tak aby umožňoval také výběr. Tato implementace se ukázala jako nejlepší, z důvodu výkonu a správné funkčnosti.

Pokud je pozice délky nula, nebo-li *min* je stejný jako *max*, tak je stále zobrazena. Její šířka, je pak velikost mezery mezi dvěma písmeny. Dále jsem musel zohlednit, jestli text má vertikální posuvník. Pokud ano, tak samotné zvýraznění musí být vertikálně posunuto v plátně, o výšku aktuálního posunu.

Backtracking je vyhodnocený stejnou funkcí, jako pro zvýraznění pozice. Jediná věc, která se liší, je forma zobrazení. Ta je ve tvaru šipky, signalizující odkud a kam se přesouvá v regulárním výrazu. Výška šipky není stejná, jako výška řádku, ale je zkrácená konkrétně na 2 pixely.

Zobrazení skupin

Skupiny jsou podobně zobrazeny, jako pozice vyhledání, nebo-li ve formě zvýraznění části textu. Jelikož skupiny mohou být vnořené, je třeba předem určit pořadí, ve kterém se budou vykreslovat. Kdyby nebyly řazeny, tak by se mohlo stát, že vnější skupina přepíše vnitřní. Pro samotné zobrazení, je potřeba měnit barvu každé skupiny, nebo zvolit jiný způsob jejich rozlišení, aby je bylo možné rozeznat. Zvolil jsem první způsob, kdy podle indexu skupiny je zvolená její barva.

6.4 Uživatelské rozhraní

Základní zobrazení je ve tmavém režimu, které lze vidět na obrázku 6.2. Rozhraní je koncipováno pouze na jednu stránku a obsahuje poměrně jednoduché ovládání. Základem rozhraní jsou dvě textová pole, kde první slouží pro zadávání regulárních výrazů a druhé pro hledaný řetězec. Oba vstupy aplikace automaticky vyhodnocuje, nicméně vstup pro hledaný řetězec čeká nějakou dobu než uživatel dopíše, aby nedocházelo k časté aktualizaci a k výslednému zpomalení aplikace.

Po dokončeném zpracování, se v pravé spodní části aplikace nachází informace, které lze vidět na obrázku 6.2. Tyto informace obsahují počet úspěšných vyhledání a počet kroků/stavů vyhledávání.

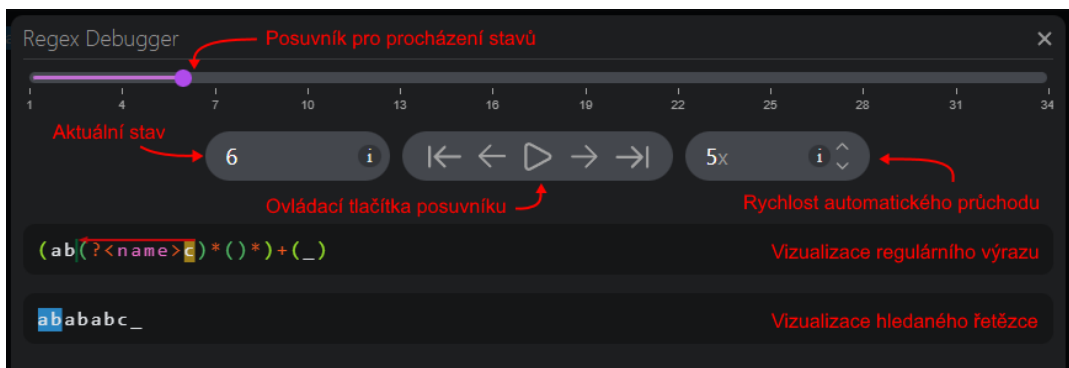
Také vedle zmíněných informací, je umístěna ikona signalizující informaci o průběhu zpracování. Ikona může být zobrazena čtyřmi různými způsoby. První je ukázán na obrázku a jedná se o celkový úspěch vyhledání. Další 3 ikony značí neúspěch, načítání resp. probíhající zpracování a poslední čekání na správně zadaný regulární výraz.

V levé spodní části aplikace je tlačítko pro otevření debuggeru. Po kliknutí na toto tlačítko se zobrazí okno, které je ukázáno na obrázku 6.3. Na vrchu tohoto okna se nachází, posuvník který slouží pro průchod stavů. S ním souvisí tři pole, které jsou přímo pod posuvníkem. Hodnota aktuálního stavu, nebo-li hodnota posuvníku se nachází v levém poli. Uprostřed je ovládání pomocí pěti tlačítek: začátek, pozice zpět, automatické přehrávání, pozice vpřed a konec. Poslední políčkem souvisejícím s posuvníkem, slouží pro manipulaci rychlosti automatického přehrávání. Rychlost je pak vyjádřena, jako $1/n$ sekund pro zobrazení nového stavu, kde n je nastavená rychlost, v případě obrázku 6.3 je $n = 5$.

Dále se v debuggeru nachází dvě pole. První slouží pro vizualizaci stavů regulárního výrazu a druhé pro vizualizaci pozice v hledaném řetězci. Stavý jsou automaticky aktualizovány, po změně hodnoty posuvníku. V regulárním výrazu v obrázku, je také zrovna vyobrazen backtracking (červená šipka zpět). V hledaném řetězci je zvýrazněná pozice, aktuálního stavu vyhledávání. Jeho součástí mohou být zobrazeny skupiny, tedy pokud již nějaké byly dokončeny.



Obrázek 6.2: Úvodní uživatelské rozhraní



Obrázek 6.3: Uživatelské rozhraní debuggeru

VSCode api zpřístupňuje využití css stylů, které má uživatel přímo nastavené ve svém prostředí. Pokud je tedy webová stránka součástí VSCode prostředí, tak přejímám styly, které má uživatel přímo nastavené ve VSCode. Součástí toho jsou, fonty, barvy, tmavý nebo světlý režim atd. Jestliže má uživatel nastavený světlý režim, tak se také stránka automaticky přizpůsobí.

6.5 VSCode rozšíření

Samotné rozšíření je vyvíjeno, pro prostředí Visual Studio Code. Výhodou tohoto prostředí, je lehká integrace webového rozhraní. VSCode API samozřejmě zprostředkovává mnohem více možností, pro samotný vývoj rozšíření.

Integrace webového zobrazení

Webové zobrazení, využívá tzv. webview, které zobrazí webovou stránku v omezeném režimu. Omezení je například ve smyslu nutností explicitně povolit JavaScriptové kódy, spouštěné v samotném zobrazení. Dalším problémem je načítání zdrojů. Ty totiž nemohou být jednoduše brány z lokálních adresářů. Musí dojít ke konverzi adresy, na adresu podporovanou VSCode. K této změně, jsem použil vyhledání všech cest ke zdrojům. Před tím než dojde k zobrazení webové stránky, je provedu konverzi na cestu podporovanou VSCode.

Zobrazení se může nacházet v několika částech prostředí. Pro tuto aplikaci jsem zvolil, že se stránka zobrazí, v pravé části na polovinu obrazovky. V budoucnu by, bylo možné přidat nastavení, aby si uživatel mohl zvolit, kde se bude okno aplikace zobrazovat.

Zobrazení aplikace lze také vyvolat najetím v textu na zadaný regulární výraz. Po najetí se zobrazí tlačítko, které po jeho stisku otevře samotnou vizualizaci. V té se bude již nacházet daný výraz, na který uživatel v textu najel. Jelikož je aplikace integrována ve vývojovém prostředí, tak je podle mě výhodné mít jako přidanou hodnotu, využití větší interaktivity s prostředím. VSCode API poskytuje možnost detekce pozice kurzoru myši, čehož jsem využil pro zmíněnou detekci. Aby bylo možné rozpoznat, zda se na pozici v textu nachází syntaxe regulárního výrazu, musí proběhnout

nějaké vyhodnocení textu. Toto vyhodnocení probíhá pomocí specifického regulárního výrazu, který zjišťuje kontrolu zda se jedná o správný tvar. Zmíněné řešení mě napadlo jako první, ale není ideální. Vhodnější by bylo použít nějaký jazykový server, který by z daného textu automaticky rozpoznal, o jaký syntaktický prvek se jedná.

Možnosti nastavení rozšíření

API pro VSCode nabízí možnost poskytnout nastavení pro rozšíření, které může uživatel měnit na základě svých preferencí. Pro tuto aplikaci jsem přidal jednoduché a omezené možnosti nastavení. První možnost nastavení se týká vypnutí a zapnutí funkce pro najetí na text, zmíněné v předchozí sekci. Druhé nastavení udává, zda se má vytvářet pokaždé nové okno webového rozhraní, nebo se má překreslovat již existující. Nastavení by mohlo existovat v budoucnu více, aby umožňovali uživatelům vyšší flexibilitu.

Komunikace rozšíření a vizualizace

Ke komunikaci VSCode s webview dochází pomocí zpráv. Jedná se o komunikaci, která je předepsána pro VSCode API součástí jejich dokumentaci. Komunikaci jsem se pokusil omezit na minimum, aby nedocházelo k přetížení aplikace. Proto se posílají pouze zprávy, které souvisí s samotným rozšířením pro VSCode. Například pokud dojde, k otevření výrazu z VSCode, tak musí dojít k jeho zaslání pomocí zprávy do webview. Vizualizace se také může nacházet samostatně mimo kontext rozšíření, proto je automaticky posílání a přijímání zpráv vypnuté.

6.6 Sestavení a spuštění projektu

Pro vlastní sestavení aplikace je potřeba mít nainstalovaný NodeJS společně s NPM. Při vývoji jsem používal NodeJS verze 20.9.0 a NPM verze 10.2.1. Avšak pro vlastní sestavení, by měla stačit verze NodeJS 14.17 společně s NPM verzí 6.14. Zároveň může být použité i novější verze. Pokud uživatel nemá NodeJS, lze jej nainstalovat ze stránky <https://nodejs.org>, NPM je součástí instalace. Lze použít i jiné balíčkové manažery místo NPM, ale v návodu na sestavení používám právě NPM. Rozšíření bylo vyvíjeno pro VSCode verze 1.84.0, ale mělo by fungovat i na novějších verzích.

V příloze A, je ukázaná částečná adresářová struktura projektu. Instalace probíhá v následujících krocích:

1. Pro instalaci všech externích závislostí, v kořenovém adresáři se použije příkaz **npm install**
2. K sestavení projektu slouží npm skript **npm run build**
3. Pro vytvoření instalačního souboru rozšíření se použije příkaz **npm run release**

4. Výsledný instalační soubory se nachází v `/libraries/regex-visualizer-extension/release`, podle přílohy A

Všechny potřebné soubory pro lokální spuštění ve webovém prohlížeči, se po přeložení nachází v adresáři `/libraries/regex-visualization/dist`. Pro spuštění je poté potřeba mít jakýkoliv webový server, kvůli podpoře web workeru.

Výsledný instalační soubor

Výsledným souborem pro VSCode rozšíření, je soubor s příponou `.vsix`. Tyto soubory slouží pro lokální instalaci rozšíření. Samotná rozšíření lze publikovat veřejně součástí VSCode marketplace, ale aktuálně jsem se rozhodl tuto aplikaci nepublikovat. Pro získání `vsix` souboru, slouží příkaz `vsce`, který dokáže sjednotit všechny závislosti do jednoho výsledného souboru.

Pro instalaci rozšíření, lze použít příkazovou paletu od VSCode. Tu lze otevřít přímo ve vývojovém prostředí, pomocí zkratky **Shift + Command + P** pro Mac, nebo pomocí **Ctrl + Shift + P** pro Windows/Linux. Po otevření příkazové palety, stačí zadat příkaz `Extensions: Install from VSIX`. Poté se otevře průzkumník souborů, kde stačí otevřít zvolený `vsix` soubor, který se nainstaluje.

Kapitola 7

Zhodnocení a testování výsledků

Pro testování při vývoji jsem používal technologii Jest, pro psaní vlastních testů. To hlavně kvůli udržitelnosti kódu a včasnému zachycení chyb při vývoji. Testy jsem používal pouze pro část aplikace, která používá složitějších algoritmů. Konkrétně při vývoji knihovny pro práci s regulárními výrazy 5. Testy jsem psal postupně podle případů, které mě napadly při vývoji. Tento přístup byl samozřejmě nedostačující, a tak jsem při testování samotné vizualizace našel několik dalších příkladů se špatnými výsledky. Ty jsem následně opravil a přidal do samotných testů.

Testování proběhlo na počítači s 6 jádrovým procesorem a 12 vlákny, AMD Ryzen™ 5 5600X. Dostupná paměť na použitém stroji, je 32GB RAM.

	Regulární výraz a testovací řetězec							
	Čas v ms							
Testovací subjekt	1a	2a	3a	4a	5a	6a	7a	7b
Regexer (vlastní knihovna)	1.936	0.672	1.061	34.205	1.004	1.556	2.382	116.117
RegExp	0.039	0.023	0.031	0.107	0.051	0.045	0.053	0.029
match	0.037	0.024	0.028	0.034	0.032	0.035	0.035	0.025
Nalezený výsledek	✓	✓	✓	✗	✓	✓	✓	✗
Násobné zpomalení Regexeru	49x	29x	34x	320x	20x	35x	45x	4000x

Tabulka 7.1: Výsledky testování výkonu zpracování regulárních výrazů

V tabulce 7.1 se nachází srovnání testovacích výsledků, vlastní implementace a standardních funkcí z jazyka JavaScript. **Regulární výrazy a testovací řetězce** vychází z přílohy B. Testovací data jsou označena číslem, které značí regulární výraz a písmenem, které odkazuje na vybraný testovací řetězec z listu B pod výrazem daným výrazem. Nalezený výsledek, značí zda testovací řetězec byl úspěšně vyhledán výrazem, či nikoliv. Násobné zpomalení *Regexeru* v tabulce znamená porovnání oproti JavaScriptové implementaci *RegExp*.

Z výsledků je patrné, že vlastní implementace je značně pomalejší a to hlavně v případě nenalezeného výsledku. Zpomalení i ve větší míře je očekávané, jelikož se jedná o implementaci psanou v

jazyce TS resp. JS. Také vlastní implementace, má za úkol vytvářet strukturu historie zpracování, což standardní implementace nemusí řešit. Velké zpomalení při neúspěchu mohlo nastat, jelikož každé zpracování probíhá pomocí NKA, ale dnešní jazyky často implementují hybridní variantu DKA s NKA a jsou velice optimalizované. DKA v tomto případě má velkou výhodu, jelikož nemusí vykonávat tolik operací, kvůli svému jednoznačnému průchodu řetězcem.

Výkonově je sice vlastní implementace *Regexer* pomalejší, ale pro účely této aplikace je tento výkon dostačující. Pro zlepšení výkonu by bylo možné přepsat tuto část aplikace do více nízkoúrovňového jazyka, jako je například *C++*, nebo *Rust*.

Kapitola 8

Závěr

Práce se zabývala tématem regulárních výrazů a možností jejich vizualizace. Její implementace byla integrována do vývojového prostředí Visual Studio Code, s tím že může existovat také mimo toto prostředí, jako samostatná webová stránka. Pro porovnání a ukázkou jsem zmínil existující nástroje pro vizualizaci regulárních výrazů, společně s jejich funkcionalitami. Následně jsem vytvořil vlastní knihovnu, která dokáže tyto výrazy zpracovávat. Přesněji daný výraz vyhodnotí a vytvoří strukturu nedeterministického konečného automatu. Ten slouží pro následující vyhledávání v zadaném řetězci. Výsledkem vyhledávání je struktura stavů, nebo-li historie procházení. Vizualizační knihovna tyto stavy přebírá a následně zprostředkovává uživatelské rozhraní pro průchod historie stavů. Rozšíření vývojového prostředí zobrazí vizualizaci do webového okna jako jeho součást.

Na konci po otestování aplikace, jsem došel k závěru že její výkon je pro své účely dostatečný. Také aplikace sice nabízí limitované množství implementovaných vzorů, ale i přesto může mít dostatečný přínos pro programátory. Aplikace byla psána tak, aby mohla být v budoucnu poměrně lehce rozšiřitelná o další funkcionality, kterými současně nedisponuje.

Každá ze tří knihoven integrovaných do aplikace, by mohla být v budoucnu rozšířená. Mezi tyto možné rozšíření, hlavně patří doplnění všech možných vzorů pro regulární výrazy z jazyka JavaScript. Další možným rozšířením aplikace, je přidání podpory vícero standardů regulárních výrazů z jiných jazyků, jako je například PCRE. Aplikace v případě nutnosti, by mohla být potencionálně zrychlená optimalizací kódu, nebo přepsáním části kódu do nízkoúrovňového jazyka, jako je C++ nebo Rust. Ve vizualizaci by bylo možné, doplnit možnost zobrazení nedeterministického konečného automatu a abstraktního syntaktického stromu daného výrazu.

Současně může být aplikace přínosná pro ostatní programátory, proto existuje možnost toto rozšíření publikovat veřejně na VSCode marketplace. Samotná publikace by mohla také urychlit případný vývoj, jelikož by existovalo více lidí, kteří by mohli aplikaci testovat. Také lze celý projekt nastavit jako veřejný, aby se na vývoji mohlo podílet více programátorů.

Literatura

1. DIB, Firas. *Build, test, and debug regex* [online]. [B.r.]. [cit. 2024-01-25]. Dostupné z: <https://regex101.com/>.
2. AVALLONE, Jeff [online]. [B.r.]. [cit. 2024-01-25]. Dostupné z: <https://regexper.com/>.
3. [online]. [B.r.]. [cit. 2024-01-25]. Dostupné z: <https://regexr.com/>.
4. AHO, Alfred V.; ULLMAN, Jeffrey D. *Chapter 10. patterns, automata, and regular expressions* [online]. 1992. [cit. 2024-04-05]. Dostupné z: <http://infolab.stanford.edu/~ullman/focs/ch10.pdf>.
5. ČERNÁ, Ivana; KŘETÍNSKÝ, Mojmír; KUČERA, Antonín. *Automaty a formální jazyky I - FI MUNI* [online]. [B.r.]. [cit. 2024-02-25]. Dostupné z: https://www.fi.muni.cz/usr/kretinsky/afj_I.pdf. Dis. pr.
6. HAVRLANT, Lukáš. *Konečný Automat* [online]. [B.r.]. [cit. 2024-02-06]. Dostupné z: <https://www.matweb.cz/konecny-automat/>.
7. VISWANATHAN, Mahesh. *Finite automata* [online]. 2017. [cit. 2024-04-11]. Dostupné z: <https://courses.engr.illinois.edu/cs475/fa2017/LectureNotes/fa.pdf>.
8. WATSON, Bruce. *A Taxonomy of Finite Automata Construction Algorithms* [online]. 1999-02 [cit. 2024-03-10].
9. XING, Guangming. *Minimized Thompson NFA*. *Int. J. Comput. Math.* [online]. 2004-09, roč. 81, s. 1097–1106 [cit. 2024-03-10]. Dostupné z DOI: 10.1080/03057920412331272153.
10. [online]. [B.r.]. [cit. 2024-02-25]. Dostupné z: <https://peggyjs.org/>.
11. *Peggyjs/peggy: Peggy: Parser generator for JavaScript* [online]. [B.r.]. [cit. 2024-02-25]. Dostupné z: <https://github.com/peggyjs/peggy>.
12. MICROSOFT. *Webview API* [online]. Microsoft, 2021 [cit. 2024-03-02]. Dostupné z: <https://code.visualstudio.com/api/extension-guides/webview#using-web-workers>.

Příloha A

Adresářová struktura projektu

```
/.....Kořenový adresář projektu
├── libraries.....Adresář vlastních knihoven
│   ├── regex-visualization.....Vizualizační knihovna
│   │   ├── customTypes.....Vlastní typy
│   │   └── src.....Adresář zdrojových kódů
│   │       ├── assets.....Zdroje např. obrázky
│   │       ├── core.....Logická část knihovny
│   │       ├── styles.....Styly aplikace
│   │       └── templates.....HTML vzory
│   ├── regex-visualizer-extension.....Knihovna rozšíření do VSCode
│   │   ├── release.....Výsledné vydání aplikace
│   │   └── src.....Adresář zdrojových kódů
│   │       ├── handlers.....VSCode handlers
│   │       ├── providers.....VSCode providers
│   │       └── web.....Adresář pro práci s webview
│   └── regexer.....Knihovna pro zpracování regulárních výrazů
│       ├── __tests__.....Kódy pro testování
│       └── src.....Adresář zdrojových kódů
│           ├── core.....Logická část knihovny
│           ├── coreTypes.....Vlastní typy
│           ├── exceptions.....Výjimky
│           └── structures.....Různé stuktury
```

Testovací data

1. **Regulární výraz:**

Testovací řetězce:

- ## 2. Regulární výraz:

$$a^*$$

Testovací řetězce:

- (a) aaa_ ✓

3. Regulární výraz:

```
(?:ab(?<name>c)*)+__
```

Testovací řetězce:

- (a) abcdabababccccabababababababcccccccc_ ✓

4. Regulární výraz:

(a|b|c)+(?:ab)+[a-z]_

Testovací řetězce:


(a) abcbbbcccbccbbccbbababababgjhrehjb_ ✗

5. Regulární výraz:

$$\wedge(a\{2,\} | b\{3\} | (x+))\{3,9\}$$

Testovací řetězce:

(a) aaxxxxxxxxxxxxxbbbxXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXaaaaabbb ✓

(b) aaaxxx 

(c) aaaxxxx ✓

(d) `axxxxxbxxxxaxxxxxbxxb` ✗

6. Regulární výraz:

$$((a|b|())+)^*|a|b|)+$$

Testovací řetězce:

(a) ababababababababababbbbababbbbababababababababbababababababbbbab-
aba ✓

(b) a ✓

7. Regulární výraz:

$$(ab(c)^*)+_{-}$$

Testovací řetězce:

(a) abcabcbccccccababababccccabcabcbccccccababababccccabcabcbccccccababababccccabcabcbccccccababababcccc
babababccccabcabcbccccccababababccccabcabcbccccccabababababccccabcabca-
bccccccabababababcccc ✓

(b) `abcabcabccccccabababababccccabcabcabccccccabababababccccabcabcabccccccabababababccccabcabcabccccccabababababccccabcabcabccccccabababababcccc` ❌