VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Vizualizace regulárních výrazů

Regular Expression Visualization

Dominik Kundra

Bakalářská práce

Vedoucí práce: Ing. Jakub Beránek

Ostrava, 2024

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Č.j.VSB/23/113247
vsboee23037B25

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 krokovat 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. Navrhněte 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

No Czech or Slovak abstract is given

Klíčová slova

No Czech or Slovak keywords are given

Abstract

No English abstract is given

Keywords

No English keywords are given

Obsah

Seznam použitých symbolů a zkratek **6**

Seznam obrázků **7**

Seznam zdrojových kódů **8**

1 Úvod **9**

2 Principy a historie regulárních výrazů **11**
2.1 Formální jazyk 11
2.2 Konečný automat 11
2.3 Bezkontextová gramatika 12
2.4 Vznik, implementace a vzory 13

3 Architektura aplikace **16**
3.1 Zvolený návrh 16
3.2 Použité technologie 17

4 Knihovna pro práci s regulárními výrazy **19**
4.1 Rozvržení 19
4.2 Parsování regulárních výrazů 21
4.3 Vyhledání pomocí regulárního výrazu 22

5 Vizualizační knihovna **28**
5.1 Návrh 28
5.2 Implementace textových editorů 28
5.3 Vizualizace průchodu 30
5.4 Uživatelské rozhraní 32

Literatura **34**

Přílohy **34**

Seznam použitých zkratek 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
HTML	–	HyperText Markup Language

Seznam obrázků

2.1 Příklad deterministického automatu přijímající slova obsahující písmena z abecedy {a, b} končící písmenem a 12
2.2 Příklad nedeterministického automatu ekvivalentního k předchozímu deterministickému 12
2.3 Převedený prázdný výraz ϵ 13
2.4 Převedený výraz **a** 14
2.5 Převedený výraz **s|t** 14

3.1 Struktura knihoven aplikace 17

4.1 Třídní diagram části knihovny pro práci s regulárními výrazy 20
4.2 Příklad výsledné JSON struktury regulárního výrazu a+ 23
4.3 NKA pro popis počítání iterací a prevenci nekonečných cyklů 24

5.1 Třídní diagram textových editorů 29
5.2 Úvodní uživatelské rozhraní 33
5.3 Uživatelské rozhraní debuggeru 33

Seznam zdrojových kódů

4.1 Počáteční neterminál 21
4.2 Výběry neterminálů, pro některé vzory regulárních výrazů 22
4.3 Uložení stavu do zásobníku 24
4.4 Vyvolání backtrackingu, pokud neexistují další přechody ze současného stavu 24

Kapitola 1

Úvod

Vyhledávání v textu patří mezi základní problémy, se kterými se velice 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**. Jedná se o sadu znaků, které nám umožňují nadefinovat výraz a ten je následně převedený do nějaké struktury. Nejčastěji jejich výsledná forma je nějaký **konečný automat**, které jsou blíže vysvětleny v kapitole 2, sekci 2.2. Téměř každý dnešní programovací jazyk je obsahuje, ale jejich implementace se mohou lišit.

Cílem této práce je na implementovat nástroj, který bude schopný procházet regulární výrazy a následně vizualizovat tyto průchody, jako rozšíření do zvoleného vývojového prostředí.

Při vývoji programů, je programátor často obeznámen s regulárními výrazy, jedná se totiž o poměrně rychlé řešení pro vyhledávání v textu. Můžeme se s nimi setkat, v podstatě skoro ve všech částech softwaru[1], např. validace formulářů, vyhledávání v textu nebo třeba v příkazovém řádku. Avšak tyto výrazy se brzy mohou stát hůře čitelnými, jelikož neumožňují v podstatě skoro žádné formátování[2]. Taktéž mohou být pro mnoho lidí matoucí, či nepřehledné. Z tohoto důvodu se hodí mít nástroj, který potencionálně usnadní práci programátorům, tak aby si mohli zobrazit průchod zadaným výrazem. Dále pro lidi, kteří například vidí tyto výrazy poprvé v životě může být snazší jim porozumět, je-li jim ukázáno jak fungují v jednotlivých krocích. Sice již existují řešení tohoto problému a to v různých formách [1, 2, 3], ale pro zvolené vývojové prostředí mnoho přístupů neexistuje. Tato situace je motivací, zabývat se problémem do hloubky 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. Abychom tohoto dosáhli musíme převést jejich textovou formu na naší chtěnou.

[1] počítačový program, aplikace
[2] upravení vzhledu, tvaru

To může být například využitím, bezkontextové gramatiky[3] nebo vlastní implementací parsování[4]. Později v kapitole 3 je vysvětleno, který ze způsobů byl zvolen a proč.

Vizualizaci regulárních výrazů můžeme chápat několika způsoby, lze si představit například zobrazení ekvivalentního konečného automatu. Další možný přístup je pomocí mapování stavů automatu do původní textové formy. Toto může být obtížné, jelikož je třeba si pamatovat nějaký vztah, mezi původním textem a formou konečného automatu. Druhý přístup byl zvolený pro tuto práci a to ve smyslu **ladícího nástroje (debugger)**. Krokování pak funguje, jako historie průchodu zadaným výrazem. Krokování je známé v programovacích prostředích jako debugger, typicky určený pro hledání chyb ve zdrojovém kódu.

Regulární výrazy pocházejí z **teoretické informatiky**[5], 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** nebo-li **grep** "Global search for Regular Expression and Print matching lines"[4].

Výsledkem této práce, je možnost procházet základní regulární výrazy ve zvoleném vývojovém prostředí. Jelikož jsou dnešní implementace velice mohutné a poskytují mnoho funkcionalit, tak není prakticky možné se zabývat celou problematikou v této práci. Proto byly vybrány prvky, které se dají obecně považovat za důležité. Těmito prvky jsou například **znaky**, **iterace**, **výběr** a **skupiny**.

V kapitole 2 jsou podrobněji popsány pojmy z teoretické informatiky, dále implementace regulárních výrazů, jejich vzory a vznik. Následně v kapitole 3 se dozvíte, jaké technologie jsou využívány a o architektuře použité v této práci. Kapitola 4 se zabývá implementací knihovny pro zpracování regulárních výrazů. Ta je pak využita v samotné vizualizaci, která je blíže popsána v kapitole 5.

[3] formální jazyk, který analyzuje a zpracovává textový řetězec
[4] syntaktická analýza textu a její přeměna na nějakou strukturu
[5] vědní obor na pomezí mezi informatikou a matematikou

Kapitola 2

Principy a historie regulárních výrazů

Tato kapitola se zabývá definicí regulárních výrazů, jak fungují a jak se jednotlivé implementace mohu lišit. Součástí jejích implementace, provází několik pojmů z teoretické informatiky, odkud pocházejí. Hlavně se jedná o **Konečné automaty** a **Thompsonovo sestrojení**.

2.1 Formální jazyk

Formální jazyk je libovolná množina konečných slov nad určitou abecedou. Slova chápeme jako řetězce znaků, která jsou přijímaná zadaným jazykem. Tato slova musí být sice konečná ale množina těchto slov může být nekonečná. Tyto jazyky mohou být definovány regulárními výrazy, formální gramatikou, konečnými automaty a dalšími. Regulární jazyky jsou pak jednou z možných definic formálních jazyků.

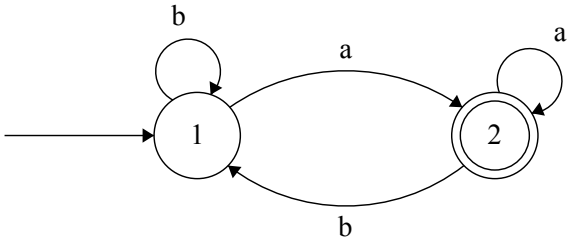
2.2 Konečný automat

Ve spojení s regulárními výrazy se často pojí konečné automaty, jedná se o další oblast v teoretické informatice. Tato práce implementuje regulární výrazy právě ve forme konečných automatů.

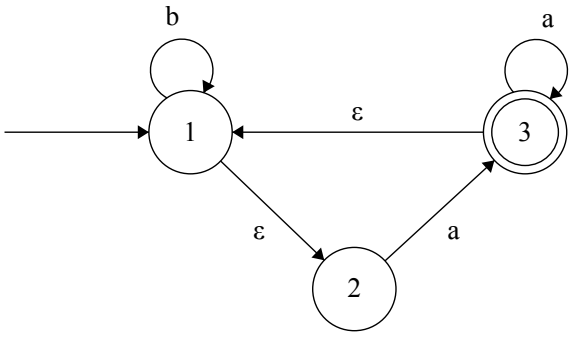
Zjednodušeně se dá říct, že konečný automat je model jednoduchého počítače, který má určitý počet stavů a přechodů [5].

Stavy jsou typicky zakreslovány jako kružnice, a každý automat musí obsahovat alespoň jeden počáteční stav, ale může jich také obsahovat více. To samé platí pro konečný/é stav/y. Konečné stavy se také vyznačují jako kruh, ale z dvojitou čárou. Počáteční stavy, jsou označovány jako stav, do kterého vede šipka, která ale nevychází z jiného stavu. Přechody jsou pak šipky vedoucí z jednoho stavu do druhého, jsou označeny přechodovým symbolem. Avšak přechod může odkazovat na stejný stav, ze kterého vychází. Tyto přechody nám říkají že pokud chceme přejít z jednoho stavu do druhého, tak musíme v přijímaném slově se posunout o daný symbol. Pokud to není možné, tak nemůžeme přejít do tohoto stavu.

Tyto 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 **maximálně** jeden přechod, dále **nemohou obsahovat tzv. prázdný znak** často označovaný řeckým písmenem epsilon ϵ. Příklad tohoto automatu je možno vidět na obrázku 2.1. NKA naopak obojí umožňují. Prázdné znaky slouží pro změnu stavu bez změny aktuální pozice v hledaném slově. Pro ukázku si můžeme porovnat dva ekvivalentní konečné automaty, NKA na obrázku 2.2 a DKA na obrázku 2.1. Nakonec je dobré podotknout, že každý NKA lze převést na ekvivalentní DKA.



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 a jelikož spadají pod teoretickou informatiku, tak si zkráceně vysvětlíme tuto oblast.

Bezkontextová gramatika, je další z možných definic formálních jazyků. Je určená konečnou množinou **neterminálních symbolů** (proměnných), konečnou množinou **terminálních symbolů**, která nesmí mít žádné prvky společné s předchozí množinou. Dále **počátečním neterminálem** s konečnou množinou **přepisových pravidel**[6].

$A \longrightarrow \beta$

kde A je neterminál a β je řetězec složený z terminálů a/nebo neterminálů. Dále šipka indikuje **přepsání** tj. levá strana se přepisuje na stranu pravou. Konečný generovaný řetězec danou gramatikou, je pouze tvořen terminálními symboly. Aby mohl být řetězec přijímaný zadanou gramatikou, musí ho být schopná tato gramatika vygenerovat.

2.4 Vznik, implementace a vzory

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

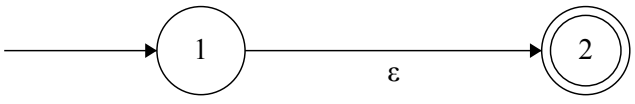
Thompsonovo sestrojení

První kdo navrhl implementaci používanou v počítačích byl **Ken Thompson**. Principem byl převod regulárního výrazu na NKA. Tato metoda se často používá v podobné či stejné formě doposud. 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 a také oproti DKA využívá **zpětného krokování (backtracking)** a povoluje složitější operace jako je **rozhlédnutí se kolem sebe (lookaround)**. Zpětné krokování je důležité pro NKA, jelikož neexistuje jednoznačná cesta vyhodnocení. DKA mají výhodu že jsou rychlejší, ale jsou typicky větší než jejich ekvivalentní NKA a neumožňují některé složitější operace. Také nepotřebují zpětné krokování, jelikož jejich cesta je deterministická tzn. existuje vždy jen jedna cesta pro hledané slovo. Někdy se ale využívá kombinace DKA i NKA, kdy DKA se využije kvůli vyšší rychlosti vyhledání daného slova a pokud bylo slovo nalezeno, tak se použije NKA pro jejich rozšířené možnosti.

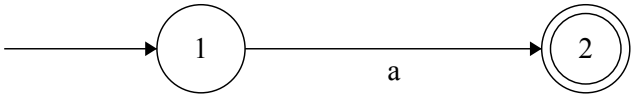
Výsledný NKA po Thompsonově sestrojení má právě jeden vstupní a výstupní stav. Thompsonovo sestrojení dále definuje následující pravidla.

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



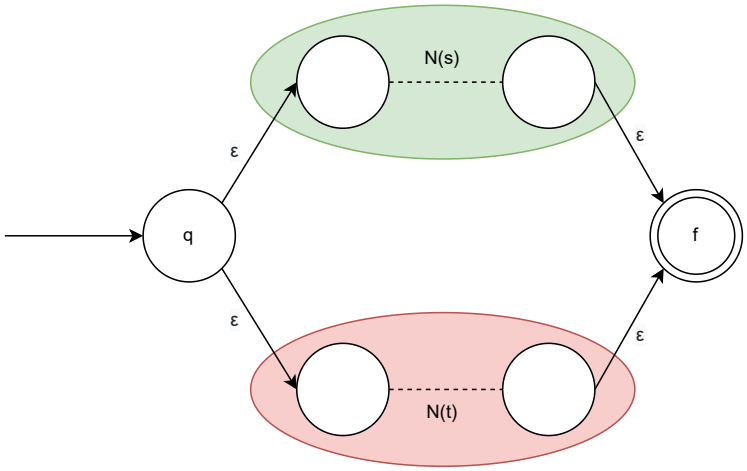
Obrázek 2.3: Převedený prázdný výraz ϵ

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



Obrázek 2.4: Převedený výraz **a**

Pro zadaný výraz **s|t**, kdy *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 jediný přechod ϵ na konečný stav *f*. Na obrázku 2.5 je vyobrazen výsledný NKA, kde skupina stavů v zelené části je *s* a červená skupina je *t*.



Obrázek 2.5: Převedený výraz **s|t**

Další pravidla pro sestrojení lze například najít v následujících článcích [7, 8].

Základní vzory

V Předchozích sekcích, již byly popsány základní konstrukce týkající se regulárních výrazů. Tato sekce se zabývá jejich základními vzory.

Za nejjednodušší výraz lze považovat prázdný výraz, také označovaný jako ϵ. Výrazy mohou obsahovat **téměř** libovolný znak, který bude přijímat slova s daným znakem. Avšak nemohou být použity znaky, které jsou rezervované, nebo-li jsou součástí syntaxe regulárních výrazů. Chceme-li použít tyto znaky je potřeba použít znak
. Takový znak se pak nazývá anglicky **escaped character**.

Iterace, je možnost jak lze opakovaně provádět nějakou operaci. Například lze iterovat znak, skupinu a další konstrukce. Prvním typem iterace je *, známa jako **Kleene star**, nebo-li kleene hvězda. Tento druh iterace může mít počet opakování od **0** až do **n** iterací, taktéž se nezívá **nula**

nebo více. Existují další 2 typy iterací a to je iterace typu **jedna nebo více** označována znakem + a **iterace v rozmezí**, která se značí {od,do}.

Operace **nebo** 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.

Implementace v programovacích jazycích

Dnes v podstatě každý programovací jazyk má v nějaké formě implementované zpracování regulárních výrazů. Tato implementace se však často liší, sice základ bývá stejný, ale rozšířená syntaxe je často odlišná. Může se tak stát to, že to co je podporované jedním jazykem, není podporované druhým. Taktéž oproti původním regulárním výrazům, dnešní implementace obsahují mnohdy složitější koncepce jako je rozhlédnutí se (Anglicky lookaround), nebo například rekurze. Někdy sice jazyky sdílí jednu stejnou funkcionalitu, ale mohou se lišit syntaxí.

Rozhlédnutí se je již celkem pokročilá funkcionalita. Jejím principem je takzvaně, nezachytávání znaků při zpracovávání. Typicky je dělíme podle směru a to na *dopředné* a *zpětné* rozhlédnutí. Pak je dělíme podle podmínění a to na *kladné* a *negativní* rozhlédnutí. 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 žádaná, jelikož pak nemusíme hledat v již nalezeném řetězci další podřetězce. Dají se dělit na zachytávající (capturing), pojmenované (named) a nezachytávající (non-capturing). Pojmenované patří pod zachytávající, akorát jsou identifikovány pomocí názvu oproti indexu. Poslední druh slouží čistě jako skupina pro regulární výraz, ale ve výsledku se neobjevuje.

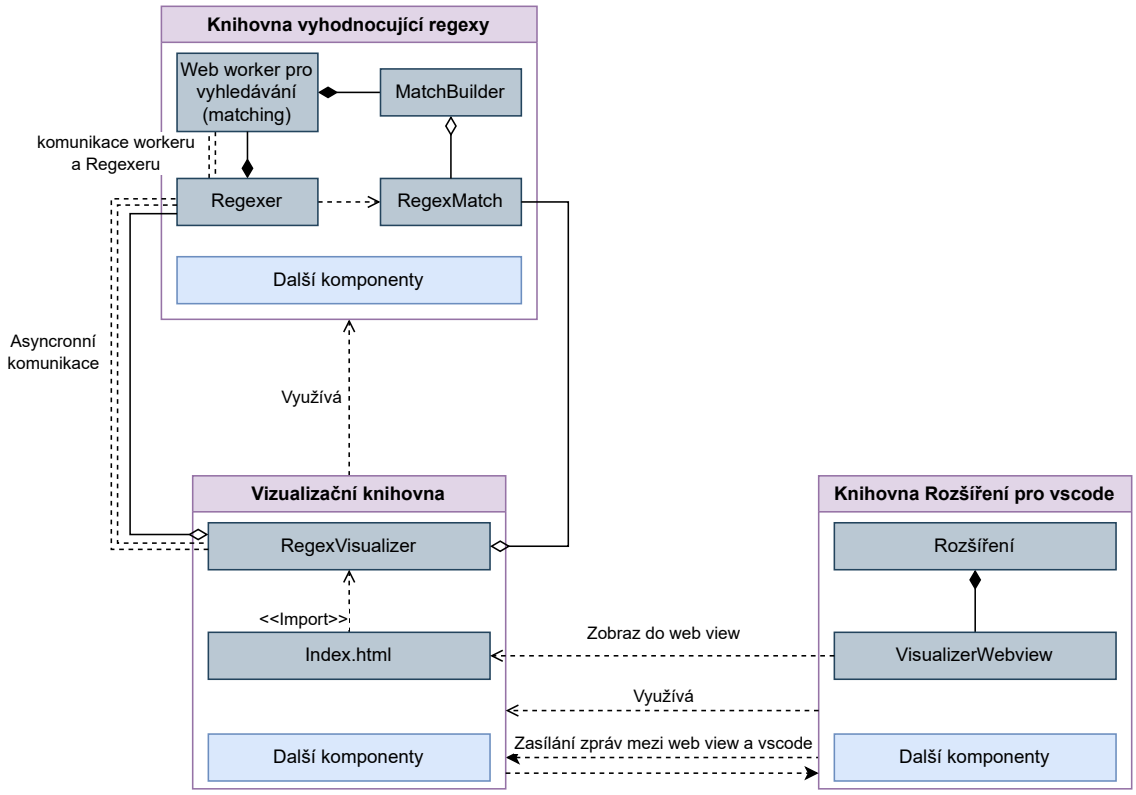
Kapitola 3

Architektura aplikace

3.1 Zvolený návrh

Aplikace je členěná na 3 základní knihovny, což je zřejmé na obrázku 3.1. Každá část má vlastní účel a jsou navzájem izolovány. První z těchto knihoven je **regexer**, ta slouží pro zpracovávání a vyhodnocování regulárních výrazů. Jako jediná z těchto knihoven může existovat plně nezávisle na ostatních knihovnách, jelikož neobsahuje závislost na žádné z dalších knihoven. Druhou knihovnou je **vizualizační**, která slouží pro samotné zobrazení zpracovávaných regulárních výrazů. Jedná se o komponentu, která může být spuštěná mimo rozšíření VSCode, například ve webovém prostředí. Tato knihovna obsahuje závislost na Regexer, ale na VSCode extension není přímá závislost, jelikož pokud není nalezená funkce pro zasílání zpráv, tak je v rámci vizualizace ignorována. Poslední částí je samotné **rozšíření**, které se stará o komunikaci s API VSCode a o řízení všeho co se týká rozšíření. Tato část aplikace implementuje vizualizační knihovnu v podobě webview (webové zobrazení).

Na obrázku 3.1 je viditelná závislost mezi knihovnami. Rovněž je patrná základní struktura těchto knihoven a jejich závislosti mezi sebou. Avšak jsou zakresleny jen ty komponenty, které můžeme považovat za nejdůležitější. Je dobré zdůraznit asynchrony komunikaci, kterou poskytuje knihovna regexer (vyhodnocující regexy). Tuto komunikaci lze vidět mezi komponentou Regexer a RegexVisualizer s tím, že Regexer využívá vedlejší vlákno mimo hlavní. Knihovna pro vlákna threads.js je popsána v sekci 3.2.



Obrázek 3.1: Struktura knihoven aplikace

3.2 Použité technologie

Tato aplikace je integrovaná do vývojového prostředí **visual studio code**, zkráceně *VSCode*. Jádro aplikace je psáno v programovacím jazyce **TypeScript**, zkráceně *TS* verze 5.3, který je nadstavbou pro jazyk **JavaScript**, zkráceně *JS*. TypeScript má jako hlavní nadstavbu, možnost využívání a přiřazování datových typů. Také platí, že kód napsaný v JS je správný pro TS, ale to neplatí naopak. Psaní větší aplikace je tak 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**[9, 10], 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 zpravovány balíčkovým manažerem **NPM** (Node Package Manager). Také využívají některých balíčků, které jsou dostupné pro npm. Část aplikace je pak postavená na technologii **NodeJS**. Jedná se o JavaScript runtime (běh programu). Například runtime VSCode rozšíření je NodeJS, oproti tomu samotné webview běží na klasickém webovém runtime, které je

typické pro webové prohlížeče.

Vizualizační část aplikace pak využívá základní **HTML** struktury. HTML je základem pro webové stránky a definuje jejich strukturu pomocí značek. Pro následnou změnu vzhledu (styl), jsem využil technologie **LESS**[6], což je rozšíření standardního **CSS**. Avšak LESS musí být transpilovaný[7] do CSS, jelikož webová stránka ho nezná. 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**[8]. 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 překladu 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, nad správností aplikace, je v algoritmické části aplikace využito technologie pro tvorbu testů. Tato knihovna se nazývá **Jest**[9]. Avšak tato knihovna slouží převážně pro testování JavaScriptových kódů, proto jsem k ní využil balíčku **ts-jest**, pro TypeScript. To mi pak umožňuje psát testy, které mohou využít typů.

Jednou z posledních knihoven, kterou jsem použil, je **Threads.js**[10]. Jelikož existují různé runtime JavaScriptu, tak neexistuje jednotný standard pro implementaci vláken (threads). Browser má tzv. *web workery* a NodeJS má *worker thready*, 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 i typy.

[6] https://lesscss.org/
[7] Typ překladu z jednoho jazyka na jazyk jiný
[8] https://webpack.js.org/
[9] https://jestjs.io/
[10] https://threads.js.org/

Kapitola 4

Knihovna pro práci s regulárními výrazy

Prvotní myšlenka, kterou jsem měl před tvorbou samotné vizualizace, byla úvaha o použití knihovny, která by mi umožňovala zpracovávat regulární výrazy. Sice implementace regulárních výrazů se nachází v samotné specifikaci JavaScriptu, ale ta mi neumožňuje získat informaci o samotném vyhledávání. Po prozkoumání, zda-li existují řešení, která by vyhovovala této práci, jsem uznal za vhodné, vytvořit vlastní implementaci v podobě této knihovny. Nenalezl jsem totiž řešení, které by bylo dostatečně flexibilní a zároveň lehce integrovatelné do programovacího jazyku TypeScript. Sice vlastní implementace je pracná, ale jelikož chci mít co nejvyšší kontrolu nad výslednou strukturou, tak je toto řešení pro tuto aplikaci asi nejvhodnější.

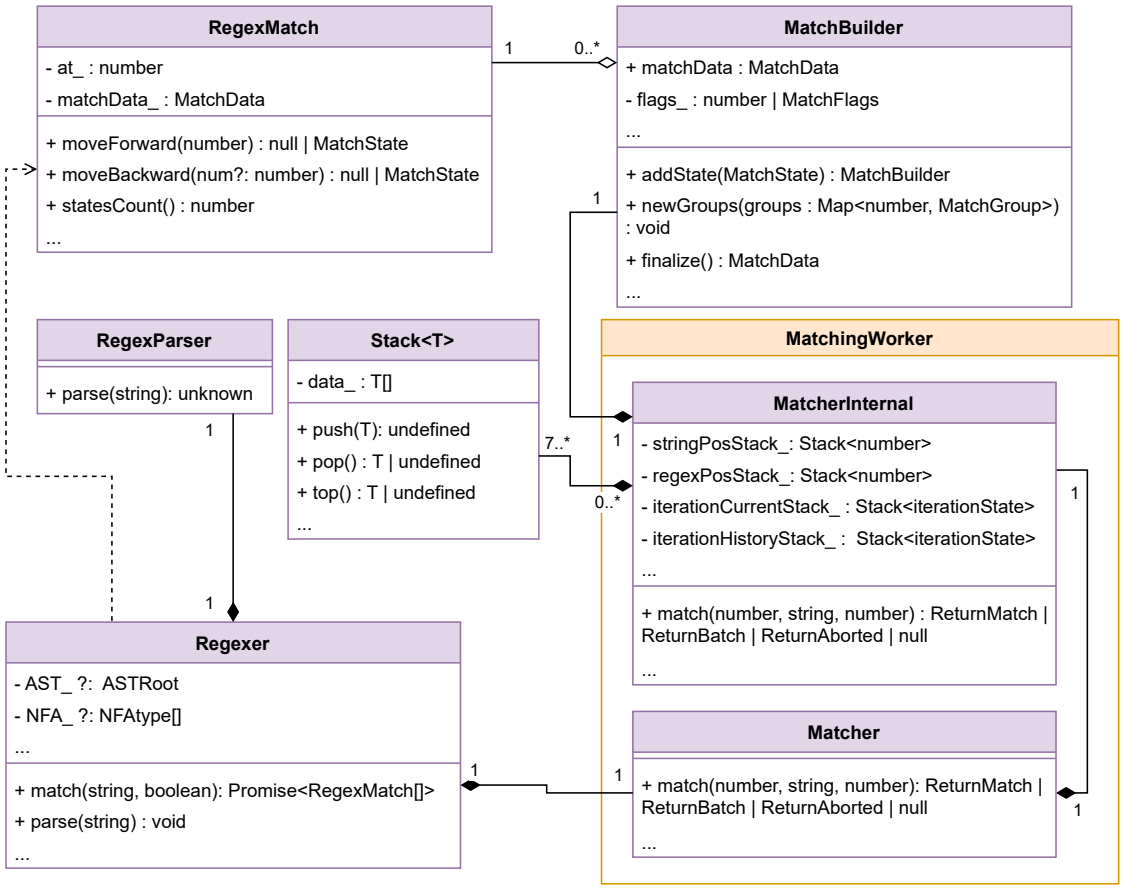
V této části práce se pokusím vysvětlit, návrh této knihovny 4.1, princip parsování regulární výrazů 4.2, jak funguje vyhledávání 4.3 a co je jeho výsledkem.

4.1 Rozvržení

Vstupní třídou, pro tuto knihovnu je **Regexer**. Slouží jako spojující a zároveň obsluhující třída. Zároveň poskytuje rozhraní této knihovny. Dále si drží důležité informace, které souvisí s aktuálním regulárním výrazem. **RegexMatch** je třída, která reprezentuje jeden výsledek vyhledávání zadaným výrazem. Její data jsou soukromá, ale umožňuje je procházet pomocí svých metod. Jedna instance teto třídy, je ekvivalentní jednomu vyhledání v textu. Data této třídy jsou generována třídou **MatchBuilder**. Instance této třídy existuje jen ve chvíli, kdy probíhá vyhledávání v zadaném řetězci. Poskytuje rozhraní, které umožňuje přidávat stavy, s tím že může tato data upravovat, dle své potřeby. Tento objekt je pak držený v třídě **MatcherInternal**, která má za úkol, průchod zadaným řetězcem, pro konkrétní výraz. Tato třída, je izolována a není dostupná z vnější, jak její název internal, v překladu vnitřní napovídá. Obsahuje hlavní logickou 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, slouží rozhraní **RegexParser**. Vzniká vždy po překladu bezkontextové

gramatiky. Jako poslední struktura, která stojí za zmínku, je **Stack**. Stack nebo-li zásobník, je velmi důležitou součástí vyhledávání. Zásobník je totiž struktura, která mi dovoluje se zbavit rekurzivního volání funkce. Rekurze obecně vede k pomalejšímu chodu programu a nelze ji jednoduše pozastavit v jakémkoliv čase. 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 daná pravidla pro zásobník. Z tohoto důvodu jsem zvolil jednoduchou implementaci zásobníku, která omezuje manipulaci s základním polem, na operace určené pro zásobník.

Vztahy mezi jednotlivými třídami, lze vidět na obrázku 4.1. Myslím si že je vhodné poukázat na obalující blok **MatchingWorker**. Jedná se o vstupní soubor vedlejšího vlákna, který slouží pro asynchronní komunikaci s hlavním vláknem.



Obrázek 4.1: Třídní diagram části knihovny pro práci s regulárními výrazy

4.2 Parsování regulárních výrazů

Parser

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

V ukázce 4.1 se nachází vstupní neterminál bezkontextové gramatiky. Ten obsahuje výběr mezi dvěma začátky. Výběr je pak dostupný pod názvem *type*, podle toho který se zvolí. Před dokončením pravidla a vracením dat, se zde může nacházet jejich modifikace. Modifikace v rámci pravidla začátku, probíhá zavoláním instance vlastní třídy **ParserHandler**, která je součástí gramatiky. Třída má za úkol zpracovávat příchozí data do struktury, která je ukázaná na obrázku 4.2.

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

Zdrojový kód 4.1: Počáteční neterminál

Existuje několik různých vzorů regulárních výrazů. Každý ze vzorů má pravidla, podle toho s jakými vzory je lze kombinovat. Ukázka kódu 4.2, obsahuje některé možné výběry pravidel, které jsem na implementoval pro parsování výrazů. Například možnosti pro iteraci, ve zmíněném kódu *to_iterate*, obsahují pouze následující vzory, které mohou být opakovány.

- Speciální znaky (*escaped_special*) – $\backslash s$, $\backslash d$
- 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 příklad **iterace** (iteration) a **výběru** (option). 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. Nebo-li byla by dříve zpracována, než-li samotný výběr. Příkladem může být výraz $a* | b*$, při kterém by se první zpracovala iterace $a*$. Výsledkem výběru by byly 2 možnosti ϵ nebo $b*$, což je sice sám o sobě správný tvar, 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 4.2: Výběry neterminálů, pro některé vzory regulárních výrazů

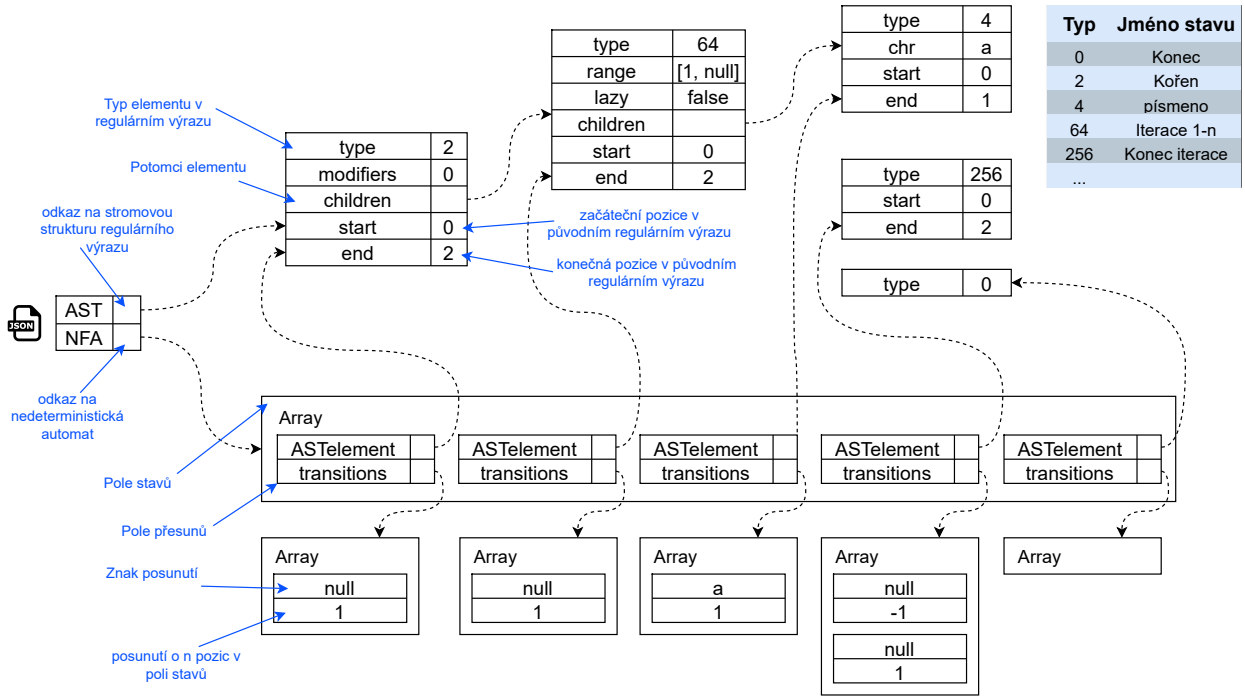
Struktura zpracovaného regulárního výrazu

Pro zpracovaný regulární výraz, jsem zvolil strukturu, která obsahuje **nedeterministický konečný automat**, zároveň s **abstraktním syntaktickým stromem** (AST), ten pak slouží k dohledání informací o původním regulárním výrazu. Výsledná struktura je datového typu, JSON (JavaScript Object Notation). JSON strukturu je možno vidět na obrázku 4.2. 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 se pak identifikuje na základě indexu v poli. Přesuny jsou pak implementovány tak, že každý stav si uchovává všechny 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 4.2 lze vidět základní JSON struktura, která obsahuje dvě vstupní hodnoty AST a NFA. Klíč NFA odkazuje na pole stavů přesunové tabulky. AST má odkaz na kořen, který signalizuje začátek regulárního výrazu. Je zde patrné, že každý stav má odkaz, na příslušící prvek v AST. AST element drží různé informace, např. pozice v původním řetězci (start a end), potomci daného stavu, nebo typ elementu. Ne každý stav musí mít potomky, ale například skupina potomky má.

4.3 Vyhledání pomocí regulárního výrazu

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



Obrázek 4.2: Příklad výsledné JSON struktury regulárního výrazu a+

Odstranění rekurze

Rekurze je sice důležitá v programování a dokáže usnadnit mnoho problémů, ale existují situace, kdy se vyplatí zbavit rekurze. V první řadě, bych rád vysvětlil, proč se vůbec rekurzivní řešení hodí, 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 jiný stav. V tuto chvíli, dojde k vrácení se v NKA do předchozích stavů a následnému vyhledávání 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 vlastních zásobníků, nebo-li program si uloží, jen potřebné informace a ve chvíli kdy dojde k vrácení se, tak vrch zásobníků se odstraní. Důležité tedy je, správně řídit správu zásobníků, což může být komplikované.

Úryvek zdrojového kódu 4.3, obsahuje základní vkládání do zásobníku, konkrétně obsahující stavy NKA a index aktuálního přesunu. Pokud se stav již nachází na vrchu zásobníku, tak je pouze navýšen index přesunu.

Příklad kódu 4.4, 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 stará o odebírání vrchů zásobníků, jako je již zmíněný zásobník stavů. Pokud není vrácená hodnota **null**, znamená to ukončení nebo pozastavení vyhledání. K ukončení dojde, pokud je zásobník stavů vyprázdněný a pozice v hledaném řetězci je na konci.

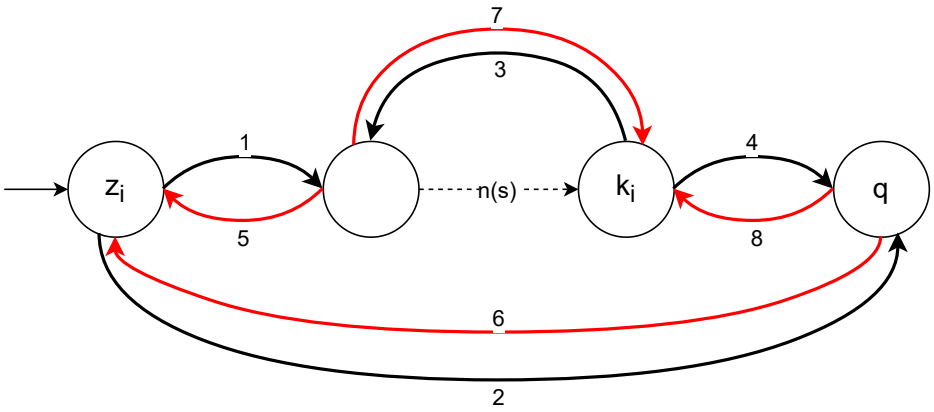
```
const nfaState = NFA[<number>this.regexPosStack_.top()] as NFAtype;
let topState = this.statesStack_.top();
if(topState?.state !== nfaState)
    this.statesStack_.push({transition: 0, state: nfaState});
else
    topState.transition++;
```

Zdrojový kód 4.3: Uložení stavu do zásobníku

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

Zdrojový kód 4.4: Vyvolání backtrackingu, pokud neexistují další přechody ze současného stavu

Počítání iterací a prevence nekonečných cyklů



Obrázek 4.3: 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 součas-**

ných iterací). Druhý značím jako Z_h (**zásobník historie**), ten slouží pro iterace, které byly již dokončené. Historie je důležitá pro backtracking.

1 – Vlož do $Z_s \longleftarrow [I_i, 0, P_s]$

2 – Vlož do $Z_h \longleftarrow [I_i, 0, P_s]$

3 – Vrchol Z_s, $I_{arr}[1] + 1$

4 – Vlož do $Z_h \longleftarrow$ 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 $Z_s \longleftarrow$ Odeber ze Z_h, $I_{arr}[1]$ - 1

Na obrázku 4.3 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 4.3, indexy pravidel korespondují s indexy v obrázku. Rád bych poukázal, že při backtrackingu se vždy zásobníky vrací, do původních stavů. To znamená, mám-li stavy a a b, tak platí pro přesun $a \rightarrow b$, při backtrackingu $a \leftarrow b$, hodnoty zásobníků v stavu a, musí být v obou případech identické. 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 nutno 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. 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. V obrázku 4.3 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 3.2. Toto zpracování dovoluje složité operace přesunout na vedlejší vlákno, aby hlavní vlákno nebylo zatíženo. Vlákna sice umožňují efektivnější rozložení 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 [11]. 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 loader, 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.

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 stav se jedná. Stav může obsahovat další data, která jsou nepovinná, nebo-li né každý stav je má. Jedná se převážně o typ akce a seznam skupin. Akce je typ 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 uložená v JSON struktuře. Ta je dále součástí třídy **RegexMatch**. Samotná třída poskytuje pouze rozhraní, pro procházení stavů, nebo popřípadě získání základních informací o vyhledávání.

Kapitola 5

Vizualizační knihovna

Tato knihovna slouží pro vizualizaci průchodu regulárním výrazem. Pro získání informací o vyhledávání, slouží již zmíněná knihovna, která byla popsána v předchozí kapitole 4. Hlavním cílem této části aplikace je, na implementovat uživatelsky přívětivé a intuitivní rozhraní. To umožňuje zadávat regulární výrazy, text ve kterém lze pomocí zadaného výrazu vyhledávat a následnou vizualizaci ve formě debuggeru.

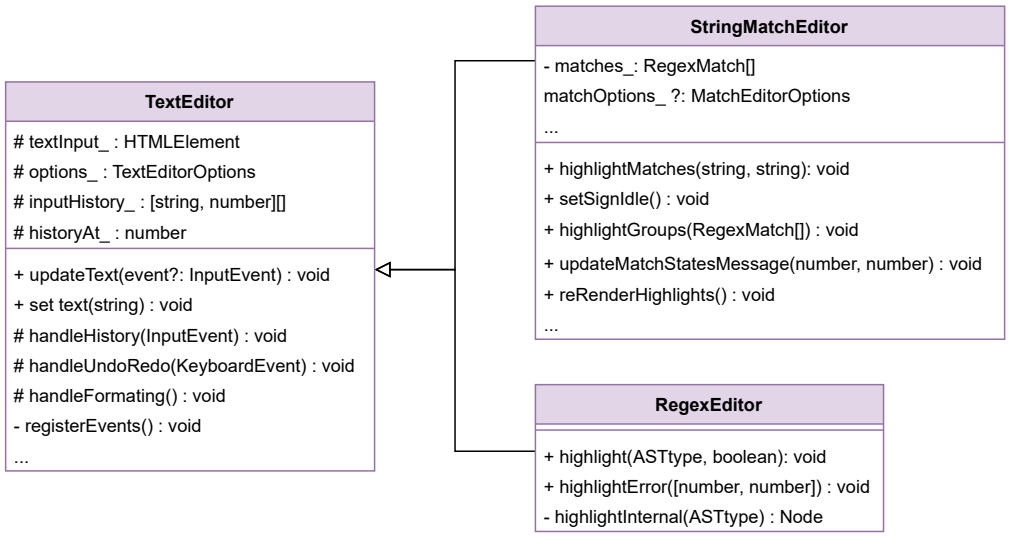
5.1 Návrh

Vstupem knihovny HTML soubor *main.html*. Ten 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* a tím 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, s informací o aktuální hodnotě posuvníku. Také má například možnost automatického přehrávání a změnu rychlosti.

Komunikace s třídou regexer

5.2 Implementace textových editorů

Ve vizualizaci se nachází dva textové editory, pro zadávání regulárního výrazu a druhý pro zadávání řetězce, ve kterém proběhne vyhledávání na základě zadaného výrazu. Jejich funkcionalita je obalena ve dvou třídách, každá sloužící pro jiný textový editor a obě dědí ze třídy *TextEditor*. Tento vztah lze vidět na obrázku 5.1.



Obrázek 5.1: Třídní diagram textových editorů

Základní řešení

Pro řešení textových editorů, jsem se rozhodl pro vlastní implementaci, pro větší flexibilitu a řešení konkrétních problému, 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 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 *handleFormating*, tak dojde ke změně podoby textu na formátovanou. Jedná se o grafické zobrazení bílých znaků, jako je nový řádek nebo tabulátor.

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 jednotlivých 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

Práci s historii jsem musel na implementovat vlastní, jelikož původní nefungovala správně. Důvodem bylo časté přepisování textu, z již zmíněného formátování. Aby historie fungovala, musel jsem vytvořit pole, které obsahuje jak původní řetězec tak pozici kurzoru v něm. Pokud byla vyvolána operace vrácení se zpět v historii (undo), překopíruje se uložený řetězec do textového pole a kurzor se nastaví na správnou pozici. K odstranění záznamu z historie nedochází, jelikož může být vyvolána *redo* operace, 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, často nastává ke změně 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 uložená 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*[11], 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 může hodit pokud je potřeba získat souřadnice písmena.

5.3 Vizualizace průchodu

Vizualizace ve formě debuggeru je obsluhována třídou *RegexDebugger*. Okno pro vizualizaci se otevře po kliknutí na tlačítko, které je předáno 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.

[11] https://stackoverflow.com/questions/69956977

Posuvník

Posuvník jsem zvolil, jako jednoduchou a intuitivní možnost procházení historie. Jeho implementace je ve vlastní třídě a jeho součástí je nastavení, pro příjemnější manipulaci 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 mít tlačítka pro ovládání, těmi jsou automatické přehrávání, vpřed, zpět, na konec a na začátek. Pro automatické přehrávání, může být součástí 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, 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. Tento koncept jsem převzal z inspirativní stránky *regex101.com*. Pozice je zvýrazněná tak, že je v pozadí pozice 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 celí řetězec a pokud byl znak součástí pozice pro zvýraznění, tak jsem rozšířil šířku zvýrazněného bloku o šířku písmene a velikost mezery. Jesli byl nalezen znak nového řádku, nebo délka zvýrazněného bloku přetekla velikost řádku, tak jsem vytvořil nový blok pro nový řádek. Problém tohoto řešení byl takový, že když došlo k nekontrolovanému zalomení řádku tzn. pokud byl zalomen řádek dříve, než konec tohoto řádku. To mohlo nastat, například pokud se zalomilo slovo. 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 uživatelskou interaktivitu.

Poslední zpusob implementace, dokáže vyřešit i zmíněný problém se zpomalením. Vychází z předchozího popisu, jelikož také využívá kurzor. 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 funkcionality.

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 scrollbar. Pokud ano, tak samotné zvýraznění musí být v plátně posunuto, o výšku aktuálního scrollu.

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 zvýraznění části textu. Jelikož skupiny mohou být vnořené, je třeba předem určit pořadí, ve kterém se budou zakreslovat. 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 rozlišení, aby je bylo možné rozeznat. Zvolil jsem první způsob, kdy podle indexu skupiny je zvolená její barva.

5.4 Uživatelské rozhraní

Základní zobrazení, je ve tmavém režimu, které lze vidět na obrázku 5.2. 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 častá aktualizace nezpomalovala aplikaci.

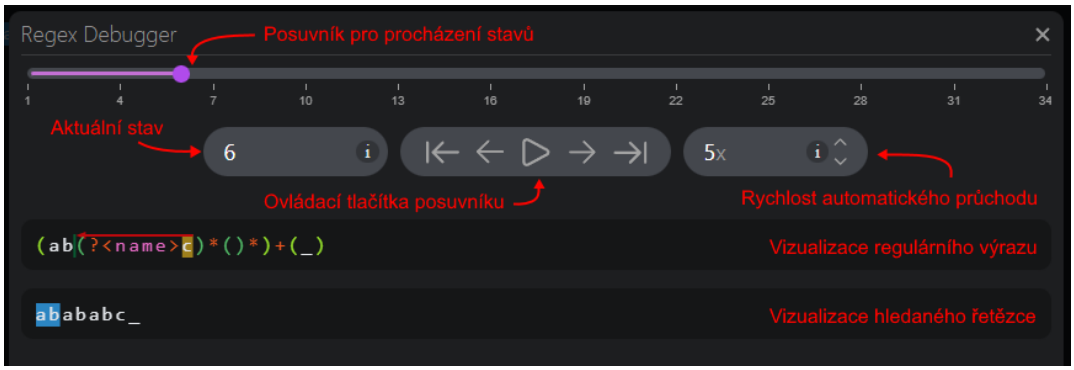
Po dokončeném zpracování, se v pravé spodní části aplikace nachází informace, které jsou vidět na obrázku 5.2. Zobrazuje se zde, kolik úspěšných vyhledání se povedlo a v kolika krocích. Také vedle zmíněných informací, je umístěna ikona signalizující informaci o průběhu zpracování. Ikona může být zobrazena čtyřmi různými způsoby. První je ukázán na obrázku a jedná se o celkový úspěch vyhledání. Další 3 ikony značí neúspěch, načítání resp. zpracovávání a poslední čekání na správně zadaný regulární výraz.

V levé spodní části aplikace je tlačítko pro otevření debuggeru, po jeho otevření se zobrazí okno, které je ukázáno na obrázku 5.3. Na vrchu okna se nachází, posuvník který slouží pro průchod stavů. S ním souvisí tři pole, které jsou pod posuvníkem. Aktuální stav, nebo-li hodnota posuvníku, se nachází v levém poli. Uprostřed je ovládání pomocí pěti tlačítek: začátek, zpět, automatické přehrávání, dopředu a konec. Poslední políčkem souvisejícím s posuvníkem, je pro manipulaci rychlosti automatického přehrávání. Rychlost je pak vyjádřena, jako $1/n$ stavů za sekundu, kde n je nastavená rychlost, v případě obrázku 5.3 je $n = 5$.

Dále se v debuggeru nachází dvě pole, pro vizualizaci stavů regulárního výrazu a druhé pro vizualizaci pozice v hledaném řetězci. Stavy jsou automaticky aktualizovány, po změně hodnoty posuvníku. V regulárním výrazu v obrázku, je zrovna vyobrazen backtracking (červená šipka zpět).

V hledaném řetězci, se zvýrazněná pozice, aktuálního stavu vyhledávání. Jeho součástí mohou být zobrazeny skupiny, pokud nějaké již byly dokončeny.

Obrázek 5.2: Úvodní uživatelské rozhraní



Obrázek 5.3: 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. Součástí toho jsou, fonty, barvy, tmavý nebo světlý režim atd. Jestliže má uživatel nastavený světlý režim, tak se stránka automaticky přizpůsobí.

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. [online]. Wikimedia Foundation, 2024 [cit. 2024-02-20]. Dostupné z: https://en.wikipedia.org/wiki/Regular_expression.
5. HAVRLANT, Lukáš. *Konečný Automat* [online]. [B.r.]. [cit. 2024-02-06]. Dostupné z: https://www.matweb.cz/konecny-automat/.
6. 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.
7. WATSON, Bruce. A Taxonomy of Finite Automata Construction Algorithms [online]. 1999-02 [cit. 2024-03-10].
8. 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.
9. [online]. [B.r.]. [cit. 2024-02-25]. Dostupné z: https://peggyjs.org/.
10. *Peggyjs/peggy: Peggy: Parser generator for JavaScript* [online]. [B.r.]. [cit. 2024-02-25]. Dostupné z: https://github.com/peggyjs/peggy.
11. MICROSOFT. *Webview API* [online]. Microsoft, 2021 [cit. 2024-03-02]. Dostupné z: https://code.visualstudio.com/api/extension-guides/webview#using-web-workers.