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 obrázků

Seznam tabulek

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 na strukturu, nejčasteji ve formě konečných automatů. Téměř každý dnešní programovací jazyk je obsahuje, ale jejich implementace se mohou lišit.

Cílem této práce je naimplementovat nástroj, který bude schopný procházet regulární výrazy a následně vizualizovat tyto průchody, jako rozšíření do 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. Tyto výrazy se však brzy mohou stát hůře čitelnými, jelikož neumožňují vpodstatě 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 snažší 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 nabídnout originální řešení v daném směru, které by mohlo být přínosem pro ostatní lidi.

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

Kapitola 2

Principy a historie regulárních výrazů

Abychom mohli porozumět, jakým problémem se v této práci vlastně zabýváme, tak si musíme zadefinovat co jsou regulární výrazy, jak fungují a jak se jednotlivé implementace mohu lišit. Následovně si vysvětlíme, jak lze zjednodušit pochopení těchto výrazů, popřípadě jak rychle najít chybu ve vlastním výrazu.

Předtím než si vysvětlíme blíže jak fungují samotné regulární výrazy, je potřeba si první objasnit význam několika pojmů z teoretické informatiky[3].

2.1 Formální jazyk

Formální jazyk je libovolná množina koečných slov nad určitou abecedou. Slova chápeme jako retězece znaků, která jsou příjimaná 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. Proto abychom pochopili proč je tato oblast pro nás důležitá, musíme si vysvětlit co vlastně jsou.

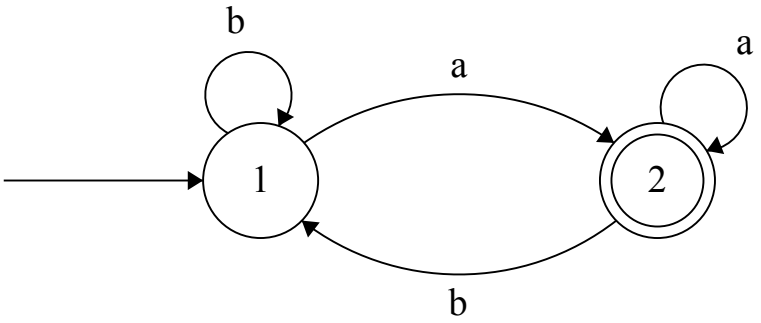
Konečný automat je model jednoduchého počítače, který má určitý počet stavů a přechodů [4].

Stavy jsou typycky zakreslovány jako kružnice, a každý automat musí obstahovat alespoň jeden počáteční stav, ale může jich také obsahovat více. To samé platí pro konečný/é stav/y. Konečné stavy se vyznačují jako kruh z dvojtou čárou a počáteční stavy jsou, označovány jako stav do kterého vede šipka, která ale nevychází z jiného stavu. Prechody jsou pak šipky vedoucí z jednoho stavu do

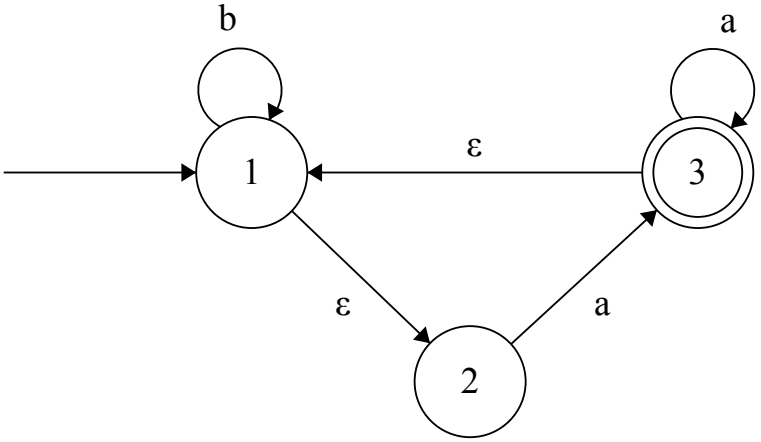
[3]vědní obor na pomezí mezi informatikou a matematikou

druhého, jsou označeny přechodovým symbolem. Tyto šipky nám říkají že pokud chceme přejít z jednoho stavu do druhého, tak musíme v příjimaném slově se posunout o daný symbol. Pokud to není možné, tak nemůžeme přejít do tohoto stavu.

Tyto automaty dělíme na deterministické a nedeterministické, zkráceně DKA (deterministický konečný automat) a NKA (nedeterministický konečný automat). DKA mohou mít v daném stavů pro každý znak abecedy **maximálně** jeden přechod, dále **nemohou obsahovat tzv. prázný znak** často označovaný řeckým písmenem epsilon ϵ. NKA naopak obojí umožňují, prázné znaky nám umožňují změnu stavu bez změny aktuální pozice v hledaném slově. Každý NKA lze převést na ekvivalentní DKA.



Obrázek 2.1: Příklad deterministického automatu příjamjící 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**[5].

$A \longrightarrow \beta$

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 příjimaný 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 byli 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

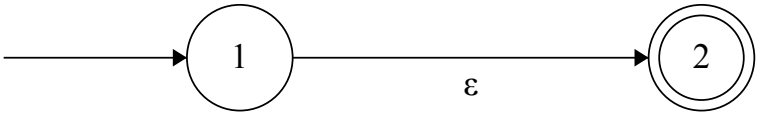
Ken Thompson byl člověkem kdo navrhnul první implementaci převodu ragulárního výrazu na NKA využívanou v počítačích, která se používá do dnes. Algoritmus se pojmenoval **Thompson's construction** (Thompsonovo sestrojení), který převádí textovou reprezentaci výrazu na ekvivalentní nedeterministický automat. Toto sestrojení je využito v této práci, proto si musíme vysvětlit jak funguje.

Používá se často implementace NKA, jelikož je poměrně jednoduchá na implementaci a také dovoluje oproti DKA využití zpětného krokování (backtracking) a rozhlédnutí se kolem sebe (lookaround). DKA mají výhodu že jsou rychlejší oproti NKA, ale jsou typycky větší než jejich ekvivalentní NKA a neumožňují již zmíněné funkce. Někdy se ale využívá například kombinace DKA i NKA, kdy DKA kvůli vyšší rychlosti se použije na vyhledání daného slova a pokud bylo nalezeno, tak se využije NKA pro jejich rozšířené možnosti.

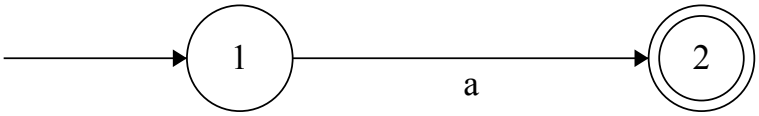
Výsledný NKA má právě jeden vstupní a výstupní stav. Aby došlo ke správnému sestrojení NKA z Regulárního výrazu, musíme se řídit několika pravidly, pro ukázku si pár těchto pravidel ukážeme.

Prázdný výraz ϵ, je převedený na vstupní stav přechod ϵ a konečný stav.

Výraz a, je převedený podobně jako prázdný výraz, ale s rozdílem přechodu a místo ϵ.

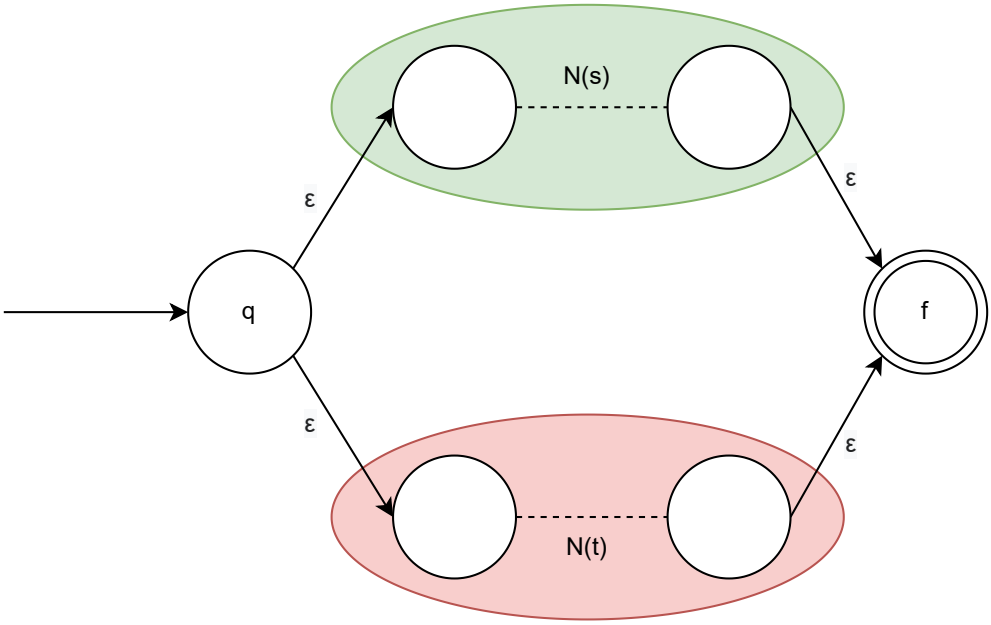


Obrázek 2.3: Převedený prázný výraz ϵ



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* 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.



Obrázek 2.5: Převedený výraz **s|t**

Další pravidla pro sestrojení lze například najít na anglické wikipedia stránce pro thompsonovo sestrojení [6].

Základní vzory

Jelikož již máme vysvětleny hlavní oblasti, které souvisí s regulárními výrazy, tak si můžeme ukázat jak vůbec vypadají a jejich základní vzory.

Jako nejzákladnější výraz můžeme považovat prázdný výraz, někdy označovaný jako ϵ. Dále výrazy mohou obsahovat **téměř** libovolný znak, který bude příjimat slova s daným znakem, avšak nemohou být použity znaky, které jsou rezervované, neboli jsou součástí syntaxe regulárních výrazů.

Iterace, je možnost jak lze opakovaně provádět nějakou operaci. Například lze iterovat znak, skupinu a další. Prvním typem iterace je *****, známa jako **Kleene star**, nebo-li kleene hvězda. Tento druh iterace může proběhnout **0** až **n** iterací, taktéž ji nazíváme **nula nebo více**. Existují další 2 typy iterací a to je iterace typu **jedna nebo více** označována znakem + a **iterace v rozmezí** {min,max}.

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.

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 osahují 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. Jejich principem je takzvaně, nezachytávání znaků při zpracovávání. Typicky je dělíme podle směru a to **dopředné** a **zpětného** rozhlédnutí. Pak je dělíme podle podmínění a to **kladné** a **negativní** rozhlédnutí, pro která platí, 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 nemusíme například, jiným regulárním výrazem hledat v již nalezeném řetězci nějaký podřetězec.

Kapitola 3

Aplikační architektura

3.1 Využíté technologie

Tato aplikace je integrovaná do vývojového prostředí **visual studio code**, zkráceně **vscode**. Jádro aplikace je psáno v programovacím jazyce **TypeScript** verze 5.3, který je nádstavbou pro jazyk **JavaScript**. TypeScript, jak z názvu vyplívá je typový JavaScript. Každý kód napsaný v JavaScriptu je správný pro TypeScript, ale to neplatí naopak. Psaní nějaké větší aplikace je tak vhodnější v TypeScriptu, čímž se můžeme vyhnout potencionálním chybám v běhu programu. Také vyvýjení rozšíření pro vscode, je možné pouze v JavaScriptu nebo TypeScriptu.

Pro parsování[4] je použita bezkontextová gramatika Peggy, pro jazyk JavaScript. Ta umožňuje poměrně snadného zpracování textové podoby regulárních výrazů do podoby strukturované. Výsledná struktura je ve formě NKA s AST (abstraktní syntaktický strom). NKA pak lze procházet a AST slouží k dohledání informací o syntaxi původního regulárního výrazu.

[4]proces kompilace a interpretace

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. HAVRLANT, Lukáš. *Konečný Automat* [online]. [B.r.]. [cit. 2024-02-06]. Dostupné z: `https://www.matweb.cz/konecny-automat/`.
5. [online]. Wikimedia Foundation, 2021-08 [cit. 2024-02-18]. Dostupné z: `https://cs.wikipedia.org/wiki/Bezkontextov%C3%A1_gramatika`.
6. [online]. Wikimedia Foundation, 2023-07 [cit. 2024-02-17]. Dostupné z: `https://en.wikipedia.org/wiki/Thompson%27s_construction`.