NPRG041 – 2023/24 ZS – Cvičení MS – **Velký úkol A01**

Regulární výrazy

Regulární výrazy

Regulární výrazy jsou jedním z prostředků, které můžeme používat k popisu tzv. regulárních jazyků. Nejenom s nimi se seznámíte v průběhu letního semestru v rámci předmětu Automaty a gramatiky. Abychom s nimi ale dokázali pracovat po praktické stránce, bez většího teoretického zázemí se určitě obejdeme.

S regulárními výrazy se setkáváme v řadě kontextů, nejenom v rámci např. programovacích jazyků nebo dotazovacích jazyků v databázových systémech. Obvykle pracujeme s variantami, které nabízí celou řadu uživatelsky přívětivých zjednodušení a zkratek, bez nich se však jsme schopni obejít. Podíváme se tedy na regulární výrazy v podobě, která je nezbytně nutná pro dosažení očekávané vyjadřovací síly.

Regulární výraz v nad nějakou abecedou Σ (např. symbolů anglické abecedy) definujeme induktivním způsobem takto. Nejprve zavedeme následující jednoduché regulární výrazy:

- $\mathtt{a}$ pro každé $\mathtt{a} \in \Sigma$ odpovídající jazyku $h(\mathtt{a}) = \{\mathtt{a}\}$, tedy jazyku, který obsahuje jeden jednosymbolový řetězec $\mathtt{a}$,
- ε (prázdný řetězec) odpovídající jazyku $h(\varepsilon) = \{\varepsilon\}$, tedy jazyku, který obsahuje jen prázdný řetězec, který značíme jako ε (někdy také λ) a
- $\emptyset$ (prázdný jazyk) odpovídající prázdnému jazyku $h(\emptyset) = \{\}$, tedy jazyku, který neobsahuje žádný řetězec.

Jsou-li r a s již definované regulární výrazy (jakkoli složité), jsme pomocí nich a následujících operací schopni zkonstruovat následující složené výrazy takto:

- $r \cdot s$ (operace $\cdot$ zřetězení) odpovídající jazyku $h(r \cdot s) = h(r) \cdot h(s) = \{uv \mid u \in h(r), v \in h(r)\}$, tedy jazyku, který obsahuje všechny řetězce vzniklé tak, že je umíme rozdělit na dva podřetězce takové, že první z nich se dá vygenerovat prvním výrazem r a druhý druhým s,
- $r + s$ (operace + alternativa) odpovídající jazyku $h(r + s) = h(r) \cup h(s)$, tedy jazyku, který obsahuje všechny řetězce, které umí vygenerovat první výraz r nebo druhý s a
- r^* (operace * iterace) odpovídající jazyku $h(r^*) = \bigcup_{i \in \mathbb{N}_0, i \geq 0} L^i$, kde $L^0 = \{\varepsilon\}$ a $L^i = L \cdot L^{i-1}$ pro $\forall\, i \in \mathbb{N}, i \geq 1$, tedy jazyku, který obsahuje všechny řetězce vzniklé tak, že je umíme rozdělit na libovolný počet podřetězců, z nichž každý se dá vygenerovat výrazem r.

Pro získání jistoty správného vyhodnocení regulárního výrazu je potřeba vhodně používat pomocné kulaté závorky. To však často není úplně nezbytně nutné, stejně tak si můžeme dovolit i další zjednodušení. My budeme konkrétně uvažovat následující dvě:

- Pokud to není nutné, závorky psát nemusíme, takže např. $((\mathtt{a}+\mathtt{b})+\mathtt{c})$ odpovídá $\mathtt{a}+\mathtt{b}+\mathtt{c}$ a $((\mathtt{a}\cdot\mathtt{b})\cdot\mathtt{c})$ odpovídá $\mathtt{a}\cdot\mathtt{b}\cdot\mathtt{c}$.
- Pokud zřetězujeme několik symbolů naší abecedy, nebudeme psát dokonce ani operátor $\cdot$ jako takový, takže např. $\mathtt{abba}$ odpovídá $\mathtt{a}\cdot\mathtt{b}\cdot\mathtt{b}\cdot\mathtt{a}$. Tuto myšlenku můžeme zobecnit tak, že po jakémkoli symbolu $\mathtt{a} \in \Sigma$ nebo zavírací závorce) nebo operátoru iterace * může bezprostředně následovat další libovolný symbol $\mathtt{a} \in \Sigma$ nebo otevírací závorka (, což ve všech těchto případech opět vede na implicitní operaci zřetězení. Např. $(\mathtt{a}+\mathtt{b})^*\mathtt{aa}(\mathtt{a}+\mathtt{b})$ odpovídá $(\mathtt{a}+\mathtt{b})^*\cdot\mathtt{a}\cdot\mathtt{a}\cdot(\mathtt{a}+\mathtt{b})$.

Abychom se ujistili, že konstrukci a významu regulárních výrazů rozumíme, pojďme se podívat na několik příkladů nad abecedou $\Sigma = \{\mathtt{a}, \mathtt{b}\}$:

- $(\mathtt{a}+\mathtt{b})^*\mathtt{abba}(\mathtt{a}+\mathtt{b})^*$ popisuje jazyk $\{w \mid w = u\mathtt{abba}v, u, v \in \{\mathtt{a}, \mathtt{b}\}^*\}$ aneb řetězce obsahující $\mathtt{abba}$
- $((\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b}))^*(\mathtt{a}+\mathtt{b})$ popisuje jazyk $\{w \mid w \in \{\mathtt{a},\ \mathtt{b}\}^*,\ |w| = 3k+1,\ k \in \mathbb{N}_0\}$, tedy jazyk obsahující řetězce s délkou dělitelnou 3 se zbytkem 1
- $\mathtt{a}(\mathtt{a}+\mathtt{b})^*\mathtt{a} + \mathtt{b}(\mathtt{a}+\mathtt{b})^*\mathtt{b} + \mathtt{a} + \mathtt{b}$ popisuje jazyk $\{w \mid w \in \{\mathtt{a}, \mathtt{b}\}^*, w$ začíná a končí na stejný symbol$\}$

Metoda sousedů

Častým problémem, který ve spojitosti s regulárními výrazy potřebujeme řešit, je situace, kdy pro daný řetězec máme rozhodnout, jestli odpovídá danému regulárnímu výrazu, tedy jestli jej dokáže vygenerovat. Např. řetězec `aba` našemu předchozímu poslednímu uvedenému regulárnímu výrazu (řetězce začínající a končící na stejný symbol) odpovídá, zatímco řetězec `abb` nikoli.

Uvedený problém je v praxi řešen pomocí tzv. konečných automatů. Takový automat však nejprve pro zadaný výraz musíme umět zkonstruovat. A k tomu slouží řada metod, např. metoda sousedů navržená Viktorem Michajlovičem Gluškovem. Abychom ji mohli použít, musíme nejprve pro zadaný regulární výraz induktivně spočítat čtveřici pomocných funkcí, na jejichž základě takový automat následně už zkonstruovat umíme. To však není naším cílem (naučíte se to až v letním semestru), tím je jen vyčíslení těchto funkcí.

Nejprve však potřebujeme ke každému výskytu nějakého symbolu abecedy $\mathtt{a} \in \Sigma$ v zadaném regulárním výrazu přiřadit pomocné číslo, označme jej jako index, pomocí kterého dokážeme navzájem rozlišit jednotlivé výskyty stejného symbolu. To se dá udělat různými způsoby, např. postupným přidělováním přirozených čísel počínaje `1`. Přesně tento postup použijeme i my. Aneb pro regulární výraz $(\mathtt{a} + \mathtt{b})^*\mathtt{ab}$ získáme jeho oindexovanou verzi jako $(\mathtt{a}_1 + \mathtt{b}_2)^*\mathtt{a}_3\mathtt{b}_4$. Pozor, ε ani $\emptyset$ nejsou symboly abecedy, a tedy ani indexy jim nepřiřazujeme.

Teď už jen musíme popsat, jaké pomocné funkce uvažujeme a jak je dokážeme pro libovolně komplikovaný oindexovaný regulární výraz r' spočítat:

- $\mathtt{Starting}(r')$ označuje množinu všech oindexovaných symbolů abecedy, na které může začínat nějaký řetězec odpovídající r'.
- $\mathtt{Neighbors}(r')$ označuje množinu všech dvojic oindexovaných symbolů, které se mohou vyskytovat bezprostředně za sebou v nějakém řetězci odpovídajícímu r'.
- $\mathtt{Ending}(r')$ označuje množinu všech oindexovaných symbolů abecedy, na které může končit nějaký řetězec odpovídající r'.
- $\mathtt{Epsilon}(r') \in \{\mathtt{true}, \mathtt{false}\}$ označuje příznak, zda r' dokáže vygenerovat prázdný řetězec ε.

Pro náš ukázkový regulární výraz $r = (\mathtt{a} + \mathtt{b})^*\mathtt{ab}$ a jeho oindexovanou verzi $r' = (\mathtt{a}_1 + \mathtt{b}_2)^*\mathtt{a}_3\mathtt{b}_4$ pak konkrétně získáme tyto hodnoty:

- $\mathtt{Starting}(r') = \{\mathtt{a}_1, \mathtt{b}_2, \mathtt{a}_3\}$
- $\mathtt{Neighbors}(r') = \{\mathtt{a}_1\mathtt{a}_1, \mathtt{a}_1\mathtt{b}_2, \mathtt{a}_1\mathtt{a}_3, \mathtt{b}_2\mathtt{a}_1, \mathtt{b}_2\mathtt{b}_2, \mathtt{b}_2\mathtt{a}_3, \mathtt{a}_3\mathtt{b}_4\}$
- $\mathtt{Ending}(r') = \{\mathtt{b}_4\}$
- $\mathtt{Epsilon}(r') = \mathtt{false}$

Chceme-li umět tyto čtyři funkce spočítat obecně, stačí postupovat podle induktivní struktury našeho regulárního výrazu. Aneb pro jednoduché výrazy vše vyjádříme triviálně, u výrazů vzniklých jednotlivými operacemi vyjdeme ze znalosti těchto funkcí u jednotlivých operandů. Ty jsou jednodušší, a tedy je opravdu vyčíslit umíme. Začněme nejprve již zmíněnými jednoduchými výrazy:

- Oindexovaný symbol $\mathtt{a}_i$, $\mathtt{a} \in \Sigma$, $i \in \mathbb{N}$
  - $\mathtt{Starting}(\mathtt{a}_i) = \{\mathtt{a}_i\}$, $\mathtt{Neighbors}(\mathtt{a}_i) = \{\}$, $\mathtt{Ending}(\mathtt{a}_i) = \{\mathtt{a}_i\}$, $\mathtt{Epsilon}(\mathtt{a}_i) = \mathtt{false}$
- Prázdný řetězec ε
  - $\mathtt{Starting}(\varepsilon) = \{\}$, $\mathtt{Neighbors}(\varepsilon) = \{\}$, $\mathtt{Ending}(\varepsilon) = \{\}$, $\mathtt{Epsilon}(\varepsilon) = \mathtt{true}$
- Prázdný jazyk $\emptyset$
  - $\mathtt{Starting}(\emptyset) = \{\}$, $\mathtt{Neighbors}(\emptyset) = \{\}$, $\mathtt{Ending}(\emptyset) = \{\}$, $\mathtt{Epsilon}(\emptyset) = \mathtt{false}$

Jsou-li r a s libovolně složité oindexované výrazy, pak pro jednotlivé operace spočítáme naše pomocné funkce takto:

- Zřetězení $r \cdot s$
  - $\mathtt{Starting}(r \cdot s) = \begin{cases} \mathtt{Starting}(r) & \text{pokud } \mathtt{Epsilon}(r) = \mathtt{false} \\ \mathtt{Starting}(r) \cup \mathtt{Starting}(s) & \text{v opačném případě} \end{cases}$

  - $\texttt{Neighbors}(r \cdot s) = \texttt{Neighbors}(r) \cup \texttt{Neighbors}(s) \cup \{xy \mid x \in \texttt{Ending}(r), y \in \texttt{Starting}(s)\}$
  - $\texttt{Ending}(r \cdot s) = \begin{cases} \texttt{Ending}(s) & \text{pokud } \texttt{Epsilon}(s) = \texttt{false} \\ \texttt{Ending}(s) \cup \texttt{Ending}(r) & \text{v opačném případě} \end{cases}$
  - $\texttt{Epsilon}(r \cdot s) = \texttt{Epsilon}(r) \wedge \texttt{Epsilon}(s)$
- Alternativa $r + s$
  - $\texttt{Starting}(r + s) = \texttt{Starting}(r) \cup \texttt{Starting}(s)$
  - $\texttt{Neighbors}(r + s) = \texttt{Neighbors}(r) \cup \texttt{Neighbors}(s)$
  - $\texttt{Ending}(r + s) = \texttt{Ending}(r) \cup \texttt{Ending}(s)$
  - $\texttt{Epsilon}(r + s) = \texttt{Epsilon}(r) \vee \texttt{Epsilon}(s)$
- Iterace r^*
  - $\texttt{Starting}(r^*) = \texttt{Starting}(r)$
  - $\texttt{Neighbors}(r^*) = \texttt{Neighbors}(r) \cup \{xy \mid x \in \texttt{Ending}(r), y \in \texttt{Starting}(r)\}$
  - $\texttt{Ending}(r^*) = \texttt{Ending}(r)$
  - $\texttt{Epsilon}(r^*) = \texttt{true}$

Zadání úkolu

Cílem tohoto úkolu je naprogramovat jednoduchou aplikaci, která pro každý regulární výraz z množiny výrazů zadané na vstupu spočítá uvedené pomocné funkce používané v metodě sousedů.

Vstupní regulární výrazy mohou být zadány dvojím způsobem: buď je najdeme přímo jako jednotlivé argumenty předané z příkazové řádky nebo je najdeme uložené v textových souborech, jejichž jména jsou opět předána formou argumentů. Pro rozlišení obou situací budeme používat dva přepínače, a to `-a` pro první způsob chování (argumenty) a `-f` pro druhý (soubory).

Předané argumenty budeme zpracovávat jeden za druhým, zleva doprava. Pokud najdeme nějaký z očekávaných přepínačů, všechny další argumenty (žádný nebo více) budeme zpracovávat příslušným způsobem (jako výraz nebo soubor), a to dokud nenajdeme další přepínač nebo už máme zpracované všechny argumenty. Pokud první argument není přepínačem, předpokládáme výchozí režim `-a`.

Za účelem zpracování těchto argumentů a nalezení všech vstupních regulárních výrazů navrhneme třídu `Reader`, její deklaraci umístíme do hlavičkového souboru `Reader.h`. Očekává se, že veškerá funkcionalita této třídy bude implementována skrze statické metody. Kromě případných dalších interních metod bude povinnou pouze následující jediná metoda:

- `static void process_arguments(const std::vector<std::string>& arguments, std::vector <std::string>& expressions)`: zpracuje předané vstupní argumenty programu a všechny nalezené vstupní regulární výrazy (z argumentů i souborů) v podobě obyčejných řetězců `std::string` uloží při zachování jejich pořadí do již vytvořeného výstupního (ne nutně prázdného) kontejneru

Vstupní soubory s regulárními výrazy budou vypadat tak, že na každém řádku bude jeden regulární výraz (a nic dalšího), případné prázdné řádky budeme přeskakovat.

Pro ilustraci se pojďme podívat na ukázkové vstupní argumenty: `(a+b) aa*+\epsilon+\emptyset* -a ab* -f file1.txt file2.txt -a (aa)*bb`. Pokud by první uvedený soubor byl zcela prázdný a druhý obsahoval dva výrazy `(aaa)*` a `a*+b*`, budou všechny nalezené regulární výrazy vypadat takto:

```
(a+b)
aa*+\epsilon+\emptyset*
ab*
(aaa)*
a*+b*
(aa)*bb
```

Všechny uvažované regulární výrazy budou nad abecedou anglických písmen (malých i velkých). Pro operace budeme používat symboly `.` (zřetězení), `+` (alternativa) a `*` (iterace), pro jednoduché výrazy prázdného řetězce použijeme namísto ε řetězec `\epsilon` a analogicky u prázdného jazyka namísto $\emptyset$ řetězec

`\emptyset`. Používat také můžeme pomocné kulaté závorky `(` a `)`. V souladu s vysvětlenými pravidly tyto závorky mohou být vynechány, stejně jako operace zřetězení. Pro tuto chvíli předpokládejme, že všechny regulární výrazy jsou korektní aneb syntakticky správně formované. Později však přidáme i schopnost detekce a korektního ošetření vybraných chybových situací.

Pro vlastní parsování regulárního výrazu použijeme rozšířený shunting-yard algoritmus, s jeho pomocí rovnou vytvoříme syntaktický strom odpovídající induktivní struktuře našeho regulárního výrazu. Pozor na to, že níže uvedený pseudokód algoritmu neřeší implicitní (chybějící) operátory zřetězení. Z hlediska operátorů pak očekáváme následující vlastnosti:

- Operátor `*` je unární postfixový a má nejvyšší precedenci
- Operátor `.` je binární infixový zleva asociativní a má střední precedenci
- Operátor `+` je binární infixový zleva asociativní a má nejnižší precedenci

---

```
1  foreach token t ve vstupním výrazu do
2      if t je symbol abecedy then
3          vytvoř pro t nový listový uzel a vlož jej do zásobníku operandů
4      else if t je otevírací závorka ( then
5          dej ( na zásobník operátorů
6      else if t je zavírací závorka ) then
7          while na vrcholu zásobníku operátorů je nějaký operátor o do
8              odeber o ze zásobníku operátorů
9              odeber dva uzly r a l ze zásobníku operandů
10             vytvoř pro o na základě l a r nový vnitřní uzel a vlož jej do zásobníku operandů
11         odeber ( ze zásobníku operátorů
12     else if t je unární postfixový operátor n then
13         odeber ze zásobníku operandů jeden uzel p
14         vytvoř pro n na základě p nový vnitřní uzel a vlož jej do zásobníku operandů
15     else t je binární infixový operátor n
16         while na zásobníku operátorů je operátor o s precedencí vyšší než n nebo také stejnou, je-li
            ovšem n současně zleva asociativní do
17             odeber o ze zásobníku operátorů
18             odeber dva uzly r a l ze zásobníku operandů
19             vytvoř pro o na základě l a r nový vnitřní uzel a vlož jej do zásobníku operandů
20         přidej n na zásobník operátorů
21 while zásobník operátorů je neprázdný do
22     odeber o ze zásobníku operátorů
23     odeber dva uzly r a l ze zásobníku operandů
24     vytvoř pro o na základě l a r nový vnitřní uzel a vlož jej do zásobníku operandů
```

Pro zapouzdření reprezentace naparsovaného regulárního výrazu navrhneme třídu `Expression`, její deklaraci umístíme do hlavičkového souboru `Expression.h`. Z hlediska veřejného rozhraní této třídy nabídneme následující konstruktor a metodu:

- `Expression(const std::string& input)`: vytvoří novou instanci regulárního výrazu na základě parsování předaného vstupního řetězce
- `Result evaluate() const`: na základě průchodu interním stromem spočítá všechny potřebné funkce z metody sousedů a vrátí jejich hodnoty v podobě instance třídy `Result`

Smyslem uvedené třídy `Result` je zapouzdřit všechny hodnoty na jednom místě, a to pomocí veřejných datových položek `std::set<Symbol> starting`, `std::set<Neighbors> neighbors`, `std::set<Symbol> ending` a `bool epsilon` pro jednotlivé funkce `Starting`, `Neighbors`, `Ending` a `Epsilon` v tomto pořadí.

Dle očekávání třída `Symbol` reprezentuje jeden oindexovaný symbol, třída `Neighbors` dvojici sousedů aneb dvojici takových oindexovaných symbolů. Obě tyto třídy musí implementovat veřejné metody `void`

`print(std::ostream& stream = std::cout) const`, prostřednictvím kterých je budeme schopni vypsat. Symbol `a` s indexem `1` vypíšeme jako `a1`, dvojici sousedů `a1` a `b2` pak vypíšeme jako `(a1, b2)`.

Pro první tři datové položky třídy výsledku jsme využili standardní kontejner množiny `std::set`, který je dostupný v knihovně `<set>`. Aby to bylo možné, je potřeba pro vkládané objekty implementovat mechanismus porovnávání, konkrétně operátor `<`. Toho snadno dosáhneme prostřednictvím globální funkce `bool operator<(const Item& left, const Item& right)`. Funkce samotná vrátí `true` právě tehdy, když první operand je ostře menší než druhý. Symboly konkrétně seřadíme vzestupně podle jejich indexů, dvojice sousedů pak primárně podle indexu prvního symbolu a sekundárně podle indexu druhého.

Za účelem korektního ošetření nejrůznějších chybových situací navrhneme vlastní hierarchii tříd výjimek. Konkrétně budeme uvažovat čtyři typy výjimek `ArgumentException`, `FileException`, `ParsingException` a `MemoryException`, všechny odvozené od společného předka `Exception`. Konstruktory nabídneme dva: `Exception(const std::string& message)` a `Exception(std::string&& message)`, kde v obou případech očekáváme textovou hlášku blíže popisující nastalou chybovou situaci. Z hlediska dalšího rozhraní už nabídneme jen metodu na získání této hlášky pomocí `const std::string& what() const`. Pokud jde o konkrétní situace vyhazování výjimek a jejich textové hlášky, předpokládáme následující chování:

- Výjimky typu `ArgumentException`
  - `Invalid option`: během zpracování vstupních argumentů najdeme neplatný přepínač (argument začínající na `-` jiný než očekávané přepínače)
- Výjimky typu `FileException`
  - `Unable to open input file`: nepodaří se otevřít uvedený vstupní soubor s regulárními výrazy
- Výjimky typu `ParsingException`
  - `Unknown token`: během parsování narazíme na nepovolený aneb neznámý token
  - `Missing operands`: nepodaří se nám sestavit uzel aktuální operace kvůli chybějícím operandům v zásobníku operandů
  - `Unmatched closing parenthesis`: při zpracování uzavírací závorky nenajdeme v zásobníku operátorů odpovídající otevírací závorku
  - `Unmatched opening parenthesis`: při závěrečném čištění zásobníku operátorů narazíme na dosud neuzavřenou otevírací kulatou závorku
  - `Unused operands`: na samotném konci algoritmu zásobník operandů obsahuje více než jeden uzel
  - `Empty expression`: nebo ve stejné situaci naopak žádný
- Výjimky typu `MemoryException`
  - `Unavailable memory`: selže dynamická alokace kvůli nedostatku paměti

Odevzdejte všechny vytvořené zdrojové soubory (`*.cpp` a `*.h`) kromě hlavního souboru `Main.cpp` s funkcí `main`. Ten předdefinovaný bude obsahovat direktivy `#include "Reader.h"` a `#include "Expression.h"`.

Cílem úkolu je ukázat schopnost práce s konstrukty, se kterými jsme se už od začátku semestru setkali. Kromě základních dovedností tedy jde o práci s argumenty programu, textovými soubory a obecně streamy, funkcemi, předávání parametrů, návrh tříd, použití konstruktorů a destruktorů, dědičnost, virtuální metody a stejně tak i dynamickou alokaci.

Předložená implementace pochopitelně musí být korektní, stabilní a bez kompilačních varování. Hodnotit se ale bude i celková kvalita kódu. Tedy zejména avšak ne výlučně organizace kódu do jednotlivých souborů, tříd a funkcí, použití hlavičkových souborů, pojmenovávání souborů, funkcí nebo i proměnných, celkový vizuální styl kódu a indentace, předávání argumentů hodnotou nebo referencí, kvalita návrhu tříd a použití dědičnosti a virtuálních metod, zbytečné neopakování stejného kódu, používání pojmenovaných konstant, ošetřování chybových situací stejně jako využívání standardních knihoven, kontejnerů nebo funkcí.