NPRG041 – 2021/22 ZS – Cvičení MS – **Domácí úkol 1**

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 dosažené 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 operaci $\cdot$ jako takovou, 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 umíme zkonstruovat. 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. přidělováním postupných čísel. 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 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 $r \cdot s$

  - $\texttt{Starting}(r \cdot s) = \begin{cases} \texttt{Starting}(r) & \text{pokud } \texttt{Epsilon}(r) = \texttt{false} \\ \texttt{Starting}(r) \cup \texttt{Starting}(s) & \text{v opačném případě} \end{cases}$
  - $\texttt{Neighbors}(r \cdot s) = \texttt{Neighbors}(r) \cup \texttt{Neighbors}(s) \cup \{xy \mid x \in \texttt{Ending}(r), y \in \texttt{Starting}(s)\}$
  - $\texttt{Ending}(r \cdot s) = \begin{cases} \texttt{Ending}(s) & \text{pokud } \texttt{Epsilon}(s) = \texttt{false} \\ \texttt{Ending}(s) \cup \texttt{Ending}(r) & \text{v opačném případě} \end{cases}$
  - $\texttt{Epsilon}(r \cdot s) = \texttt{Epsilon}(r) \wedge \texttt{Epsilon}(s)$
- 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 úkolu je naprogramovat jednoduchou aplikaci, která pro množinu regulárních výrazů na vstupu spočítá uvedené pomocné funkce používané v metodě sousedů a vypíše je v určeném formátu na standardní výstup.

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).

Jednotlivé předané argumenty budeme zpracovávat jeden za druhým, zleva doprava. Pokud najdeme nějaký z uvedených přepínačů, všechny další argumenty (žádný nebo více) budeme zpracovávat příslušným způsobem (argument/soubor), a to dokud nenajdeme další přepínač nebo už jsme zpracovali všechny argumenty. Pokud první argument není přepínačem, předpokládáme výchozí režim `-a`. Všechny uvedené soubory budou vypadat tak, že na každém řádku bude jeden regulární výraz (a nic dalšího), případné prázdné řádky budeme přeskakovat.

Pro ilustraci, vstupní argumenty mohou vypadat např. takto: `(a+b) aa*+\epsilon+\emptyset* ab* -f file1.txt file2.txt file3.txt -a (aa)*bb`.

Všechny regulární výrazy na vstupu budou nad abecedou anglických písmen. 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. Můžeme předpokládat, že všechny regulární výrazy jsou korektní aneb syntakticky správně formované.

Pro vlastní parsování regulárního výrazu použijte rozšířený shunting-yard algoritmus, s jeho pomocí rovnou vytvořte syntaktický strom odpovídající induktivní struktuře našeho regulárního výrazu. Na základě průchodu tímto stromem spočítejte (a ve vhodné podobě ponechte uložené u každého uzlu) všechny potřebné funkce `Starting`, `Neighbors`, `Ending` a `Epsilon`. Pozor, níže uvedený pseudokód algoritmu neřeší implicitní (chybějící) operátory zřetězení. Z hlediska operátorů očekávejte 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

```
inicializuj generátor indexů na hodnotu 0
foreach token t ve vstupním výrazu do
    if t je číslo then
        inkrementuj o 1 hodnotu generátoru indexů na i
        vytvoř pro t na základě i nový listový uzel a vlož jej do zásobníku operandů
    else if t je otevírací závorka ( then
        dej ( na zásobník operátorů
    else if t je zavírací závorka ) then
        while na vrcholu zásobníku operátorů je nějaký operátor o do
            odeber o ze zásobníku operátorů
            odeber dva uzly r a l ze zásobníku operandů
            vytvoř pro o na základě l a r nový vnitřní uzel a vlož jej do zásobníku operandů
        odeber ( ze zásobníku operátorů
    else if t je unární postfixový operátor n then
        odeber ze zásobníku operandů jeden uzel p
        vytvoř pro n na základě p nový vnitřní uzel a vlož jej do zásobníku operandů
    else t je binární infixový operátor n
        while na zásobníku operátorů je operátor o s precedencí vyšší než n nebo také stejnou, je-li
          ovšem n současně zleva asociativní do
            odeber o ze zásobníku operátorů
            odeber dva uzly r a l ze zásobníku operandů
            vytvoř pro o na základě l a r nový vnitřní uzel a vlož jej do zásobníku operandů
        přidej n na zásobník operátorů
while zásobník operátorů je neprázdný do
    odeber o ze zásobníku operátorů
    odeber dva uzly r a l ze zásobníku operandů
    vytvoř pro o na základě l a r nový vnitřní uzel a vlož jej do zásobníku operandů
zásobník operandů nyní obsahuje jediný uzel, a to kořenový uzel celého stromu
```

Vstupní regulární výrazy zpracujte ve stejném pořadí, v jakém jste je objevili na vstupu, respektujte tedy pořadí předaných argumentů a pořadí výrazů uvnitř souborů. Pro každý výraz vypište výstup v následujícím formátu:

```
(aa+bb)*
- Starting: {a:1, b:3}
- Neighbors: {(a:1, a:2), (a:2, a:1), (a:2, b:3), (b:3, b:4), (b:4, a:1), (b:4, b:3)}
- Ending: {a:2, b:4}
- Epsilon: true

ab*a
- Starting: {a:1}
- Neighbors: {(a:1, b:2), (a:1, a:3), (b:2, b:2), (b:2, a:3)}
- Ending: {a:3}
- Epsilon: false
```

Každý blok pro jeden konkrétní regulární výraz ukončíte jedním dalším prázdným řádkem (mezi bloky a za posledním tedy bude vždy jeden volný řádek). Regulární výraz jako takový vypíšete přesně v té podobě, v jaké jste jej dostali na vstupu (aneb neprogramujte vlastní funkci, která by výraz dokázala vypsat vhodným průchodem syntaktického stromu, ale opravdu přímo využijte výraz ze zadání). Oindexované symboly u funkcí `Starting` a `Ending` seřaďte vzestupně podle indexů těchto symbolů. Dvojice sousedních symbolů u funkce `Neighbors` seřaďte primárně podle indexu prvního symbolu a sekundárně podle indexu druhého. Funkce `Epsilon` vypíše `true` nebo `false`.

Pro implementaci funkcí `Starting`, `Ending` i `Neighbors` použijte standardní kontejner pro množinu, tedy kontejner `std::set` dostupný v hlavičkovém souboru `<set>`. Aby to bylo možné, je potřeba pro vkládané objekty implementovat porovnávání, konkrétně operátor `<`. Toho snadno dosáhneme např. prostřednictvím funkce `bool operator<(const Symbol& left, const Symbol& right);`, kde `Symbol` je třída, která naše objekty reprezentuje.

Pokud veškerá činnost celého programu proběhne bez problémů, vrátí aplikace kód `0`. V případě jakékoli chyby pak kód `1`. V takovém případě obsah na standardním výstupu není podstatný. Připomeňme, že vstupní regulární výrazy budou vždy korektní, v jiných případech ale případné chyby (např. během práce se soubory) už ošetřit potřeba je.

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 a stabilní. 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, deklarace významných konstant, ošetřování chybových situací stejně jako využívání standardních knihoven, kontejnerů nebo funkcí.

Odevzdejte jen zdrojové soubory (*.cpp, *.h, *.hpp). Za úkol je možné získat až 15 bodů, každý započatý týden zpoždění bude penalizován 5 body.