Jméno: **UČO:**

0007 *list* 1 *učo* *body*

Oblast strojově snímaných informací. Své učo a číslo listu vyplňte zleva dle vzoru číslic. Jinak do této oblasti nezasahujte. 0123456789

1. [1 bod] Tento příklad je poněkud netradiční v tom, že vaším úkolem bude vytvořit program. Konkrétně si vyzkoušíte implementaci regulárních výrazů, tak jak je máme definované v tomto předmětu (o vztahu k regulárním výrazům v běžných programovacích jazycích se zájemci dozví na konci zadání). Kostru pro programování naleznete v interaktivní osnově.

Regulární výrazy

Připomeňme si, jak vypadá regulární výraz (definice 2.58 ve skriptech):
Množina regulárních výrazů nad abecedou Σ, označovaná $RE(\Sigma)$, je definována takto:

1. ε, $\emptyset$, a a pro každé $a \in \Sigma$ jsou regulární výrazy nad Σ.
2. Jsou-li E, F regulární výrazy nad Σ, pak také $(E \,.\, F)$, $(E + F)$ a (E^*) jsou regulární výrazy nad Σ.
3. Každý regulární výraz vznikne po konečném počtu aplikací kroků 1–2.

V praxi samozřejmě chceme moci závorky vynechávat, to pro nás ale nebude podstatné. Budeme se totiž zabývat jen překladem z interní stromové reprezentace regulárních výrazů (viz dále) na *rozhodovač příslušnosti*, tedy něco, co nám pro dané slovo dokáže rozhodnout, zda je regulárním výrazem akceptované či nikoli.

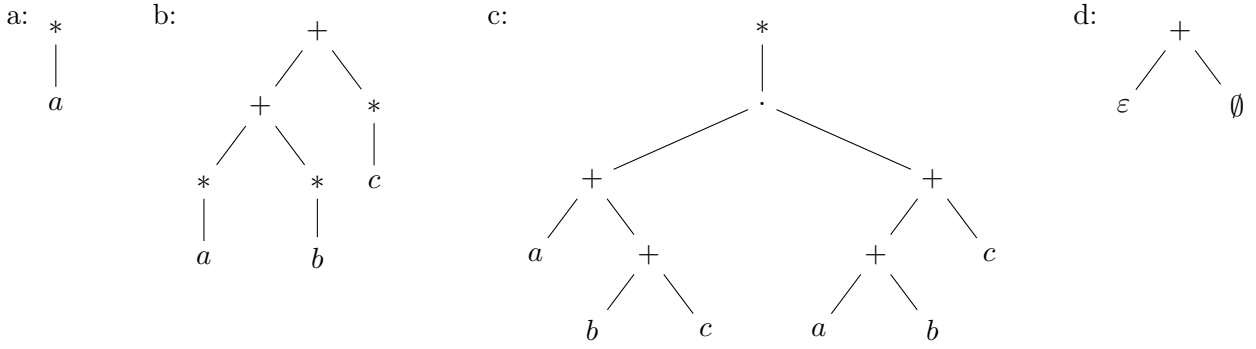
Zaveďme si rovnou několik příkladů, s nimiž budeme pracovat (všechny nad $\Sigma = \{a, b, c\}$):
(a) (a^*)
(b) $(((a^*) + (b^*)) + (c^*))$
(c) $\big(\big((a + (b + c)) \,.\, ((a + b) + c)\big)^*\big)$ – jazyk všech slov sudé délky
(d) $(\varepsilon + \emptyset)$

Reprezentace regulárních výrazů

Regulární výraz (jako v podstatě libovolnou strukturu definovanou induktivně nebo pomocí bezkontextové gramatiky) můžeme reprezentovat stromem. Ten bude v tomto případě mít v listech znaky ze Σ, ε, nebo symbol prázdné množiny a v uzlech operace.

Pro naše příklady vypadají stromové reprezentace následovně:

a: b: c: d:



Reprezentace regulárních výrazů v Pythonu

Úkol budeme programovat v Pythonu. Popišme si tedy reprezentaci (stromové struktury) regulárních výrazů v Pythonu. Nejprve budeme potřebovat importovat knihovny (pro typové anotace a definice výčtových typů):

Jméno: **UČO:**

0007 *list* 2 *učo* *body*

Oblast strojově snímaných informací. Své učo a číslo listu vyplňte zleva dle vzoru číslic. Jinak do této oblasti nezasahujte. 0123456789

```
from typing import Union, List, Optional
import enum
```

Následně si definujeme typ pro reprezentaci typu uzlu ve stromě.

```
class Operation(enum.Enum):
    Epsilon = enum.auto()     # reprezentuje typ listu, který je prázdné slovo
    Emptyset = enum.auto()    # list, který je prázdná množina
    Literal = enum.auto()     # list, který je znak
    Union = enum.auto()       # sjednocení (+)
    Concat = enum.auto()      # zřetězení (.)
    Iteration = enum.auto()   # iterace (*)
```

Zdědění z typu `enum.Enum` nám umožňuje relativně elegantně definovat výčtový typ – tedy typ, který má pouze vyjmenované hodnoty (viz komentáře v definici). Při použití je třeba před hodnotu přidat název typu, např. `Operation.Union`.

Nyní již samotná stromová reprezentace regulárního výrazu:

```
class Regex:
    def __init__(self,
                 op_or_value: Union[str, Operation],
                 left: Optional["Regex"] = None,
                 right: Optional["Regex"] = None) -> None:
        if isinstance(op_or_value, str):
            # if the first arg is a string we have a literal
            self.op: Operation = Operation.Literal
            self.value: Optional[str] = op_or_value
            assert len(self.value) == 1
        else:
            # otherwise we have an operator
            self.op = op_or_value
            self.value = None
        self.left = left
        self.right = right
```

Budeme předpokládat, že konstruktor třídy `Regex` se buď volá s jednoznakovým řetězcem nebo s operací (parameter `op_or_value`). Pro operaci může následovat nula, jeden nebo dva regexy podle arity operace (parametry `left` a `right`). Pro reprezentaci slova ε a prázdného jazyka $\emptyset$ používáme pouze typ operace – `Regex(Operation.Epsilon)` a `Regex(Operation.Emptyset)`. Nevalidní regexy nemusíte uvažovat, nebudou testovány.
Naše regulární výrazy tedy můžeme reprezentovat jako:

```
re_a = Regex(Operation.Iteration, Regex("a"))

re_b = Regex(Operation.Union,
             Regex(Operation.Union,
                   Regex(Operation.Iteration, Regex("a")),
                   Regex(Operation.Iteration, Regex("b"))),
             Regex(Operation.Iteration, Regex("c")))
```

Jméno: **UČO:**

0007 *list* 3 *učo* *body*

Oblast strojově snímaných informací. Své učo a číslo listu vyplňte zleva dle vzoru číslic. Jinak do této oblasti nezasahujte. 0123456789

```
re_c = Regex(Operation.Iteration,
             Regex(Operation.Concat,
                   Regex(Operation.Union,
                         Regex("a"),
                         Regex(Operation.Union, Regex("b"), Regex("c"))),
                   Regex(Operation.Union,
                         Regex(Operation.Union, Regex("a"), Regex("b")),
                         Regex("c"))))

re_d = Regex(Operation.Union,
             Regex(Operation.Epsilon),
             Regex(Operation.Emptyset))
```

Poznámka: Abeceda je v našem případě dána implicitně – budou to právě ty znaky, které se objevují v literálech daného regulárního výrazu.

Zadání implementace

Vaším úkolem bude implementovat třídu `CompiledRegex`, která představuje rozhodovač příslušnosti, a funkci `compile`. Jedinou povinnou metodou třídy `CompiledRegex` je funkce `match`, beroucí řetězec a vracející `bool`, který vyjadřuje, zda vstupní slovo patří do jazyka reprezentovaného příslušnou hodnotu `CompiledRegex`. Pokud slovo obsahuje znak, který nepatří do abecedy daného regulárního výrazu, pak musí být zamítnuto. Funkce `compile` bere `Regex` a vrací `CompiledRegex` reprezentující daný jazyk.

```
class CompiledRegex:  # TODO
    def match(self, value: str) -> bool:
        ...

def compile(regex: Regex) -> CompiledRegex:
    ... # TODO
```

Můžete si přidávat libovolné další globální funkce, třídy a další metody do třídy `CompiledRegex`. Dále je v podstatě jedno, jak svou funkcionalitu rozložíte mezi `compile` a `match`. Předpokládané použití je nicméně takové, že `compile` se zavolá pro daný regulární výraz jednou a následně se používá (potenciálně mnohokrát) funkce `match`. Tato funkce pro dané slovo rozhodne, zda jej původní regulární výraz akceptuje nebo ne.

Pro získání plného počtů bodů je třeba, aby funkce `match` byla *efektivní*, tedy rozpoznávala slova rychle. Prozraďme, že optimální implementace `match` má složitost $\mathcal{O}(|w|)$, kde w je vstupní slovo. O něco složitější (ale stále pro nás efektivní) implementace má složitost $\mathcal{O}(|w| \cdot |r|)$, kde r je daný regulární výraz. V obou případech zanedbáváme složitosti případných vyhledávacích struktur, jako jsou slovníky.

Příklad

```
cre_a = compile(re_a)
cre_a.match("aaa")    # → True
cre_a.match("")       # → True
cre_a.match("b")      # → False
cre_a.match("ab")     # → False
```

```
cre_b = compile(re_b)
cre_b.match("aaa")    # → True
cre_b.match("bb")     # → True
cre_b.match("aabb")   # → False
cre_b.match("bca")    # → False
```

Jméno: **UČO:**

0007 *list* 4 *učo* *body*

Oblast strojově snímaných informací. Své učo a číslo listu vyplňte zleva dle vzoru číslic. Jinak do této oblasti nezasahujte. 0123456789

```
cre_c = compile(re_c)
cre_c.match("")       # → True
cre_c.match("a")      # → False
cre_c.match("abca")   # → True
cre_c.match("abbba")  # → False
```

```
cre_d = compile(re_d)
cre_d.match("")       # → True
cre_d.match("a")      # → False
cre_d.match("abca")   # → False
cre_d.match("abbba")  # → False
```

Hodnocení Podmínkou pozitivního hodnocení je projití testy za splnění časových limitů. Časový limit se vztahuje samostatně na `compile` a `CompiledRegex.match`. U obou dbejte na rozumnou efektivitu, hlavní část testů složitosti je však u `CompiledRegex.match`.

- Za funkčnost na krátkých vstupech (slova do 10 znaků) můžete získat 0,3 bodů.
- Za funkčnost i na dlouhých vstupech (řádově až stovky znaků vstupu, regulární výrazy mohou být též velké) můžete získat dalších 0,7 bodů.

Technické požadavky

- Do odevzdávárny odevzdávejte jediný soubor `.py` s vaší implementací. Typy `Operation` a `Regex` nesmíte měnit, stejně tak nesmíte měnit rozhraní funkcí `compile` a `CompiledRegex.match`.
- Smíte používat pouze moduly `typing`, `enum`, `collections` a `unicodedata`. Pokud byste rádi použili nějaký další modul, napište s dostatečným předstihem do diskusního fóra zdůvodnění, proč byste jej chtěli, posoudíme to. Nebudeme nicméně povolovat žádné moduly týkající se regulárních výrazů či parsování v Pythonu (tedy jistě ne modul `re`).
- **Máte 5 možností odevzdání**, počítá se výsledek z nejlepšího odevzdání.
- Výsledky se zobrazí v poznámkovém bloku úkolu 0701 do několika minut (typicky do 20).
  - Pokud ale bude mnoho lidí odevzdávat těsně před deadline, pak je možné, že dojde k zahlcení systému a vyhodnocování bude trvat déle – na případné pomalejší vyhodnocování nebudeme brát zřetel.
  - Rozhodujícím časem z hlediska splnění termínu je čas vložení do odevzdávárny.
- Každý nový, přepsaný nebo přejmenovaný soubor v odevzdávárně je potenciálně vyhodnocen a stojí vás jeden pokus – dávejte si tedy pozor, abyste souborů nevložili více najednou či je nepřejmenovávali. Nespoléhejte se na to, že v případě chyby stihnete soubor smazat.
- Typová kontrola nemusí projít, nicméně silně doporučujeme si ji před odevzdáním spustit, protože vám může pomoci odhalit chyby (`mypy --strict reseni.py`).
- Před ostrými testy se spustí testy odpovídající příkladům v tomto zadání (a funkci `main` v kostře), pokud tyto elementární testy neprojdou tak se odevzdání nepočítá.
- Příklad musíte vypracovávat samostatně, bez sdílení kódu mezi sebou a bez přebírání kódu z internetu (myšlenku algoritmů přebírat můžete, implementaci však nikoli).
- Na vyhodnocovacím serveru je Python 3.9.2, řešení tedy musí fungovat v něm.
- Případné dotazy směřujte jako vždy do diskusního fóra.

Jméno: **UČO:**

0007 *list* 5 *učo* *body*

Oblast strojově snímaných informací. Své učo a číslo listu vyplňte zleva dle vzoru číslic. Jinak do této oblasti nezasahujte. 0123456789

Regulární výrazy v programování

Asi jste si už všimli, že regulární výrazy máme i v programovacích jazycích. Typickými jazyky známými pro použití regulárních výrazů jsou Perl, AWK nebo sed[1], ale regulární výrazy najdete například i v Pythonu, C++, C# nebo Javě a s pomocí knihoven je lze používat i v C.

Tyto regulární výrazy se ale zapisují jinak než ty v IB005 a kromě rozhodování příslušnosti dokáží také označovat předem určená podslova či provádět substituce. Dále se však tyto regulární výrazy často liší i svou vyjadřovací silou. Například v Pythonu je výraz `ww = re.compile(r"([ab]*)\1")` schopný rozpoznávat jazyk $L = \{ww \mid w \in \{a, b\}^*\}$. Tento jazyk nejen že není regulární, není dokonce ani bezkontextový.

Kromě toho, že „regulární“ výrazy v programovacích jazycích často nepokrývají jen regulární jazyky, se rovněž často liší efektivitou. Právě díky pokročilým vlastnostem často tyto „regulární“ výrazy backtrackují a mohou tedy mít exponenciální časovou složitost. Tyto problémy v mnoha jazycích (včetně Pythonu a Perlu) přetrvávají i u „regulárních“ výrazů, které se omezují jen na operace z našich formálních regulárních výrazů.

[1] sed typicky známe jako utilitu schopnou filtrovat text a provádět v něm substituce pomocí regulárních výrazů, jeho programovací jazyk je však turingovsky úplný, teoreticky v něm tedy lze řešit libovolný algoritmicky řešitelný problém.