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

2. [1 bod] Tento příklad je programovací. Vyzkoušíte si na něm parsování (analýzu) jednoduchého bezkontextového jazyka. Kostru pro programování naleznete opět v interaktivní osnově.

Začneme gramatikou pro náš jazyk:

$$\Sigma = \{\underline{\emptyset}, \underline{a}, \underline{b}, \underline{c}, \underline{\varepsilon}, \underline{(}, \underline{)}, \underline{[}, \underline{]}, \underline{+}, \underline{.}\}$$
$$G = (\{\langle Start\rangle, \langle Literal\rangle, \langle BOp\rangle\}, \Sigma, P, \langle Start\rangle)$$

$$\begin{aligned} P = \{ \quad \langle Start\rangle &\to \underline{\emptyset} \mid \langle Literal\rangle \mid \underline{(}\ \langle Start\rangle\ \langle BOp\rangle\ \langle Start\rangle\ \underline{)} \mid \underline{[}\ \langle Start\rangle\ \underline{]}, \\ \langle Literal\rangle &\to \underline{a} \mid \underline{b} \mid \underline{c} \mid \underline{\varepsilon}, \\ \langle BOp\rangle &\to \underline{+} \mid \underline{.}\ \}. \end{aligned}$$

Tato gramatika nám popisuje jazyk regulárních výrazů nad abecedou $\Phi = \{a, b, c\}$ s tím, že místo abychom iteraci řešili postfixovou hvězdičkou, tak na ni máme hranaté závorky (to nám usnadní parsování). Zároveň si všimněte, že výrazy jsou plně uzávorkované a nepotřebujeme tedy řešit prioritu operátorů, což nám výrazně zjednoduší práci. Nicméně i bez těchto zjednodušení by byl jazyk bezkontextový.

Derivační a abstraktní syntaktické stromy

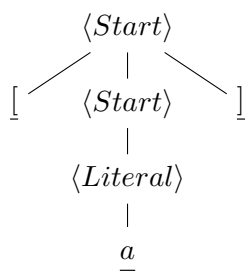
Jak jistě víte, odvození slova z gramatiky lze popsat derivačním stromem daného slova. Při parsování bychom v podstatě mohli chtít zbudovat tento derivační strom,[1] v praxi nám ale bude stačit (a dokonce bude praktičtější) o něco jednodušší *abstraktní syntaktický strom* (zkráceně budeme používat AST z anglického *abstract syntax tree*).

Abstraktní syntaktický strom abstrahuje od specifik zápisu slova která nejsou podstatná pro jeho zpracování (v našem případě jsou slova regulární výrazy). Například při parsování jazyka C můžeme zapomínat opakující se bílé znaky (mimo řetězce). Typicky též můžeme abstrahovat od závorek určujících prioritu operátorů – tyto závorky máme totiž zachycené ve struktuře stromu. Můžeme též zjednodušit strukturu stromu – například sloučit řetězce neterminálů. Obecně formát AST závisí především na potřebách dalšího zpracování.

Ukažme si nyní několik příkladů derivačních a abstraktních syntaktických stromů pro regulární výrazy. Regulární výrazy máme vždy uvedené tak, jak je zapisujeme v tomto předmětu (s dodatečnými závorkami) a následně jako slova našeho jazyka $\mathcal{L}(G)$.

a. $(a^*) \approx \underline{[a]}$

Derivační strom:



AST:

(v AST jsme hranaté závorky převedli na vyjádření, že jde o iteraci)

[1] Některé nástroje na automatické generování parserů z gramatik, například ANTLR, to tak i dělají.

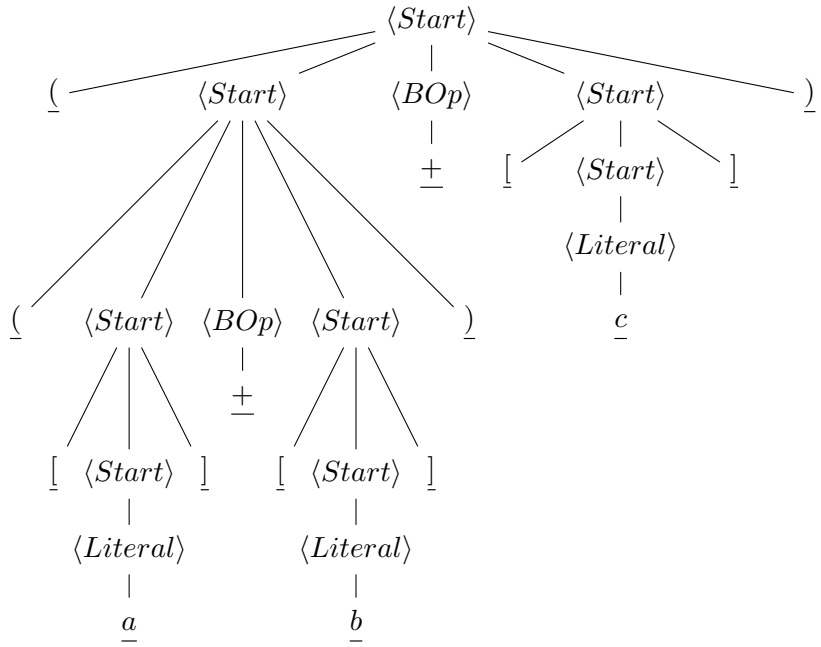
Jméno: **UČO:**

0007 *list* 2 *učo* *body*

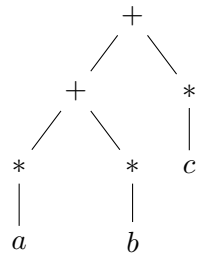
Oblast strojově snímaných informací. Své učo a číslo listu vyplňte zleva dle vzoru číslic. Jinak do této oblasti nezasahujte. 0123456789

b. $(((a^*) + (b^*)) + (c^*)) \approx \underline{(([a] \underline{+} [b]) \underline{+} [c])}$

Derivační strom:

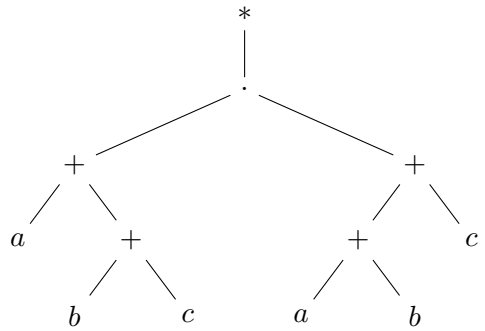


AST:



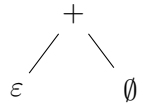
c. $\big((a + (b + c)) \,.\, ((a + b) + c)\big)^* \approx \underline{[((a \underline{+} (\underline{b \underline{+} c})) \,\underline{.}\, (\underline{(a \underline{+} b)} \underline{+} c))]}$

AST:



d. $(\varepsilon + \emptyset) \approx \underline{(\varepsilon \underline{+} \emptyset)}$

AST:



Jak jste si jistě všimli, abstraktní syntaktický strom regulárního výrazu v našem případě odpovídá právě stromové reprezentaci, kterou jsme používali v příkladu 07.01.

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

Zadání implementace

Výstupní formát tohoto příkladu je právě vstupní formát příkladu 07.01 – výstupem tedy bude hodnota typu `Regex`, definici této třídy naleznete v kostře zadání a její popis případně v zadání příkladu 07.01. Tím však propojení končí, žádná návaznost na implementaci z příkladu 07.01 tu tedy není.

Vaším úkolem je naprogramovat jedinou funkci `parse`, která vezme řetězec a vrací odpovídající hodnotu typu `Regex`, pokud je regulární výraz správně utvořený, nebo hodnotu typu `int` udávající pozici (index) první chyby, pokud není správně utvořený.

```
def parse(str_regex: str) -> Union[Regex, int]:
    ...  # TODO
```

Vaše funkce `parse` musí produkovat AST odpovídající přímo vstupu, bez jakýchkoli optimalizací – takové AST je pro každý platný zápis regulárního výrazu právě jedno, budeme tedy testovat přesnou shodu. Dále nesmíte nijak rozšiřovat jazyk gramatiky – mohlo by se vám pak snadno stát, že akceptujete něco, co budeme testovat jako nevalidní vstup.

Pro zpřehlednění zápisu budeme umožňovat psát mezi libovolné terminály libovolné množství mezer, při parsování je budeme ignorovat.[2]

Pozice chyby Pro nesprávně utvořené regulární výrazy vraťte index první pozice, která je nutně chybou. Tím myslíme takový index, že prefix vstupu končící na daném indexu nelze nijak doplnit na validní regulární výraz. Index se klasicky počítá od 0, do pozice se počítají i mezery. Například pro výraz `"(a)"` je chybový index 2, protože koncová závorka je nutně špatně (kulatá závorka může být jen kolem binární operace). Obdobně pro výraz `"(a ]"` je chybový index 3, protože na něm je ukončující hranaté závorka namísto binární operace (kterýkoli předchozí index nemůže být chybový index, protože výraz lze stále doplnit na validní výraz, například `"(a + b)"`). Jako speciální případ považujeme pro účely reportování chyb konec vstupu za *virtuální znak* nacházející se za posledním reálným znakem vstupu. Například tedy `"(a"` má chybu na indexu 2, výraz `""` má chybu na indexu 0. Pokud nezvládnete splnit tuto část úkolu, můžete se ztrátou 0,2 bodu odevzdat řešení, které vrací libovolnou hodnotu typu `int` jako indikaci chyby.

Příklady vyhodnocení[3]

```
parse("")          # → 0
parse("\u03B5")    # → Regex(Operation.Epsilon)
parse("\u2205")    # → Regex(Operation.Emptyset)
parse("(a + b)")   # → Regex(Operation.Union, Regex("a"), Regex("b"))
parse("(a + x)")   # → 5      # x není povolený literál
```

[2] Formálně vzato bychom je měli mít v gramatice, fakticky by ji to ale znepřehlednilo a nic dobrého pro implementaci by to nepřineslo.

[3] Python plně podporuje unicode znak a proto budeme v řetězcích ε a $\emptyset$ zapisovat odpovídajícími znaky, LaTeX a jeho fonty bohužel unicode tak dobře nepodporují, proto máme tyto znaky v příkladech tu zapsané pomocí jejich ekvivalentní zápisu entitami (jde pouze o jiný zápis, nikoli o jinou reprezentaci v Pythonu).

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

```
parse("( ([a] + [b])  + [c])")
    # → Regex(Operation.Union,
    #         Regex(Operation.Union,
    #               Regex(Operation.Iteration, Regex("a")),
    #               Regex(Operation.Iteration, Regex("b"))),
    #         Regex(Operation.Iteration, Regex("c")))
parse("[a)")       # → 2
parse("a + b")     # → 2     # chybí kulaté závorky
parse("(a)")       # → 2     # kulaté závorky mohou být jen kolem binární operace
parse("[a + b]")   # → 3     # chybí kulaté závorky
parse("aa")        # → 1     # dva literály za sebou
```

Opět platí, že je v podstatě jedno, jaký přístup k parsování zvolíte, musíte ale splnit podmínky na efektivitu. Na základě tohoto předmětu byste měli být schopni odvodit si hned několik algoritmických postupů, které pro naši jednoduchou gramatiku povedou k efektivnímu řešení (byť obecně až tak efektivní nejsou, o obecně efektivních postupech pojednává například předmět IA006 Vybrané kapitoly z teorie automatů).

Efektivita V tomto příkladě požadujeme aby parsování probíhalo v lineárním čase vzhledem k délce vstupu (opět zanedbávaje případná vyhledávání ve slovnících a podobné operace, které sice nejsou asymptoticky konstantní, ale v praxi je za konstantní můžeme často považovat). Doporučujeme parsovat na jeden průchod vstupního řetězce.

Další omezení Mohlo by se snad zdát, že při parsování by šlo použít regulární výrazy. Je ale třeba si uvědomit, že zadaný jazyk není regulární a tudíž regulární výrazy nejsou dobrý nástroj na jeho parsování. Aby vás to tedy nelákalo, nebudete mít povoleno používat pythonovský modul `re` pro zpracování regulárních výrazů. Teoreticky byste to sice mohli obejít použitím svého řešení 07.01, je to ale silně nedoporučené.

Zakázáno je rovněž používat různé nástroje na generování parserů či knihovny pro psaní parserů – podobně jako u regulárních výrazů v příkladu 07.01 nám jde o to, abyste si vyzkoušeli na jednoduchém příkladu základy, na nichž tyto nástroje stojí, nikoli abyste se jen naučili používat nějaký konkrétní nástroj.

Hodnocení

Podmínkou pozitivního hodnocení je projití testy za splnění časových limitů.

- Za funkčnost na krátkých vstupech (slova do 16 znaků) můžete získat **0,3 body**. Jakýkoli `int` je považován za korektní indikaci nevalidního vstupu.
- Za funkčnost i na dlouhých vstupech můžete získat dalších **0,5 bodů**. Jakýkoli `int` je považován za korektní indikaci nevalidního vstupu.
- Za správné vracení indexu chyby můžete získat další **0,2 body**.

Hloubka reprezentace výsledných regulárních výrazů bude ve všech případech taková, aby bylo možné je s dostatečnou rezervou procházet rekursivně i s ohledem na limity rekurse v Pythonu.

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

Technické požadavky

- Do odevzdávárny odevzdávejte jediný soubor `.py` s vaší implementací. Typy `Operation` a `Regex` nesmíte měnit, stejně tak nesmíte měnit rozhraní funkce `parse`.
- Smíte používat pouze moduly `typing`, `enum`, `collections`, `unicodedata`, `__future__` a `math`. 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 11.02 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.

Praktická poznámka – lexikální analýza

V praxi se u „velkých“, typicky programovacích, jazyků používá i regulární část parsování, která běží před samotnou bezkontextovou analýzou. Tato část se nazývá lexikální analyzátor (neboli *lexer*) a její úlohou je rozdělit vstupní řetězec na tokeny, které pak slouží v podstatě jako terminály bezkontextové syntaktické analýzy.[4] Tyto tokeny typicky odpovídají „slovům“ vstupního jazyka odděleným mezerami či dalšími oddělovači (závorkami apod.) – například `if`, `else`, `foo.bar`. Výhodou tohoto přístupu je, že analyzátor se pak již nemusí zabývat vstupem na úrovni znaků, ale na úrovni tokenů a je tedy o něco jednodušší a přehlednější. Náš jazyk je nicméně natolik jednoduchý, že by nám takové rozdělení nic nepřineslo (mimo jiné proto, že nemáme žádné terminály skládající se z více znaků).

[4] Oproti klasickým terminálům ale mohou mít nějakou další strukturu, například token „identifikátor“ může obsahovat jméno identifikátoru.