Schelet si checker pentru fiecare limbaj:

- skel-py-e2.zip
- skel-scala-e2.zip

Deadline etapa 2: 12 decembrie ora 23:59

Proiect

Etapa 2

Etapa 2 consta in **parsarea** unei expresii regulate (regex) scrisa in maniera conventionala, si conversia acesteia in forma prenex. [1)]

Forma standard a expresiilor regulate

Forma standard a regex-urilor poate fi descrisa in forma BNF (https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form) astfel:

```
<regex> ::= <regex><regex> |
            <regex> '|' <regex> |
            <regex>'*' | <regex>'+' | <regex>'?' |
            '(' <regex> ')' |
            "[A-Z]" |
            "[a-z]" |
            "[0-9]" |
            "eps" | <character>
```

In descrierea de mai sus, elementele dintre parantezele angulare <> sunt **non-terminali** care trebuie generati, caracterele sunt intotdeauna plasate intre ghilimele simple, iar sirurile intre ghilimele duble.

`<character>` se refera la orice caracter obisnuit care nu face parte din caractele de *control* (precum `*` sau `|`), sau la orice sir de lungime trei de forma `'c'` , unde `c` poate fi orice caracter inclusiv de control.

“eps” reprezinta caracterul Epsilon.

Preprocesarea Regex-urilor

In descrierea de mai sus, pe langa caracterele alfa-numerice si operatiile de baza star, concat si union, veti gasi si:

1. doua operatii noi:
    a. plus `+` - expresia asupra careia este aplicat apare de 1 data sau mai multe ori.
    b. semnul intrebarii `?` - expresia asupra careia este aplicat apare o data sau niciodata.
2. 3 syntactic sugars:
    a. `[a-z]` - orice caracter litera mica din alfabetul englez
    b. `[A-Z]` - orice caracter litera mare din alfabetul englez

c. `[0-9]` - orice cifra

Aceste operatii noi nu contribuie la expresivitatea regex-urilor, insa ajuta foarte mult utilizatorii sa scrie regex-uri compacte si usor de citit. In implementarea voastra, este recomandat sa *preprocesati* regexurile, adica sa eliminati operatorii nou-introdusi si sa ii inlocuiti cu cei standard. Operatorii standard sunt cei prezentati la curs (concatenare, reuniune si star).

Spre exemplu: $e+ = ee*$ sau $[0-9] = 0 \cup 1 \cup 2 \cup \ldots \cup 9$.

In felul acesta, AST-ul va avea un numar minimal de **tipuri** de noduri, iar algoritmul Thompson cat mai putine cazuri diferite de tratat.

Caractere de control sau obisnuite?

O problema care a aparut deja inclusiv la etapa 1 are legatura cu rolul caracterelor intr-un regex. Caracterele pot fi *de control* (precum `()*|` dar si *whitespace*) sau obisnuite. Insa dorim sa folosim caractere de control si cu rolul de caractere obisnuite. In acest caz, acestea trebuie intotdeauna *escapate* folosind ghilimele. Spre exemplu, nu putem folosi spatii albe intr-un regex decat *escapat* - `' '`.

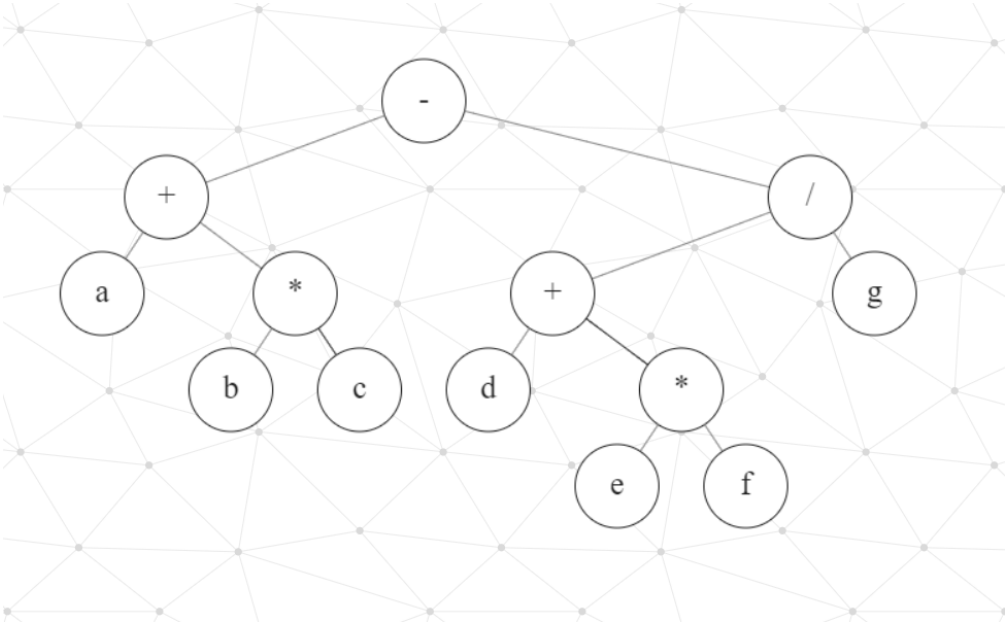
In acelasi timp, in cadrul parsarii, este important sa stim rolul pe care il are un caracter citit (de control sau obisnuit). Pentru a reprezenta aceasta diferenta in Python, aveti in schelet doua clase: `Character`, `Operator`. In Scala, puteti folosi tipul de date `Either[A,B]` avand constructorii `Left(v:A)` si `Right(v:B)`.

Precedenta

Avantajul formei Prenex este ca folosirea parantezelor nu mai este necesara pentru a specifica prioritatea operatiilor. In forma standard trebuie insa sa avem grija la prioritatea operatiilor pentru a evalua corect o expresie regulata.

Facem o scurta analogie cu ordinea operatiilor aritmetice: `+` si `-`, `*` si `/`, respectiv paranteze pentru a intelege mai usor ordinea operatiilor din Regex-uri.

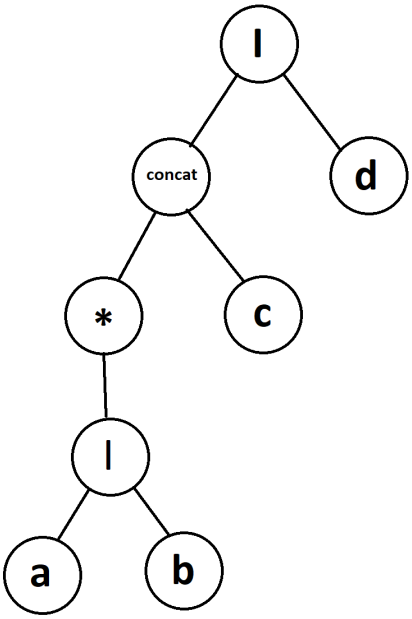
Prioritatea operatiilor aritmetice este: paranteze `()`, inmultiri sau impartiri (`*` sau `/`), adunari sau scaderi (`+` sau `-`). Astfel, expresia `a + b*c - (d + e*f)/g` se va transforma in urmatorul AST:



Putem observa ca paranteza se evalueaza inaintea inmultirilor, iar inmultirile inaintea adunarilor.

Similar, prioritatea pentru Regex-uri este paranteze `()`, star `*`, concat (nu are un simbol asociat) si union `|`.

De exemplu `(a|b)*c|d`, care va genera arborele:



Din arbore putem genera usor forma prenex: `UNION CONCAT STAR UNION a b c d`.

Implementare

Implementarea consta in parsarea unui Regex si transformarea sa in forma Prenex. Pentru acest lucru se recomanda folosirea unui AST - Abstract Syntax Tree. Puteti folosi exact AST-ul implementat la Etapa 1, adaugand o metoda de afisare (eventual chiar `toString`) pentru a obtine forma prenex).

Parsarea expresiilor prenex

Pentru a parsa un Regex, avem nevoie de o stiva care sa retina parti ale expresiei / operatii parsate deja. Vom interactiona in doua feluri cu stiva:

- reducerea expresiilor (sau cooling):
  - Exemplul 1: daca pe stiva avem: `0 | Star(?) | …` , atunci vom inlocui cele doua expresii cu : `Star(0) | …`
  - Exemplul 2: daca pe stiva avem: `0 | 0 | Concat(?,?) | …` , rezultatul reducerii va fi: `Concat(0,0) | …`
- adaugarea expresiilor: vom citi operatorul sau operandul curent, si vom adauga elementele corespunzatoare pe stiva.

Trebuie sa avem grija la ordinea operatiilor pentru a putea traduce expresia corect: de exemplu, daca intalnim operatorul * (star), ar trebui sa facem un cooling, fiind un operator unar cu cea mai mare prioritate.

Implementarea voastra trebuie sa combine in mod eficient adaugarea cu reducerea.

Testare

Testarea este similara cu cea de la etapa 1. Mai mult, este necesara implementarea intregii etape 1, deoarece vom testa comportamentul corect al DFA-ului construit si nu rezultatul transformarii. Astfel, la etapa 2 veti obtine o forma prenex dintr-un Regex: Regex → Prenex, pe care o sa il dam mai departe in transformarea facuta la etapa 1 Prenex → NFA → DFA (→ MinDFA eventual). La final testam daca DFA-ul construit accepta/respinge un set de cuvinte.

Verificarea corectitudinii implementarii voastre se va face automat, printr-o serie de teste unitare, o parte punctate si o parte nepunctate. Aceste teste nu acopera fiecare caz posibil si testeaza doar comportarea corecta a AFD-urilor obtinute din cateva expresii regulate, pe cateva secvente reprezentative.

Sunteti incurajati sa va adaugati propriile teste:

1. pentru a asigura corectitudinea pe mai multe cazuri simple sau intermediare
2. pentru a testa alte componente intermediare ale codului vostru (de exemplu parsarea corecta a expresiilor in forma prenex si construirea unui arbore corect pentru acestea).

O abordare eficienta, economica dpdv al timpului, de scris cod poate fi sumarizata astfel:

1. scriem un test pentru o functie/componenta noua, sau o parte bine determinata a acesteia (e.g. parsarea corecta a reuniunii a doua regexuri)
2. scriem implementarea pentru acea componenta
3. folosim eventuale afisari **doar** pentru debugging, atunci cand nu este evident de ce un test pica
4. cand un test trece, trecem la urmatoarea componenta
5. cand e necesar sa modificam componente la care am lucrat anterior, re-rulam toate testele anterioare, pentru a ne asigura ca modificarea nu a afectat corectitudinea codului

Folderul care contine testele va fi suprascris de checker, testele luate in considerare pentru nota fiind doar cele din skeletul de cod.

Python

Pentru rularea testelor folositi comanda `python3 -m unittest` . Aceasta comanda va detecta automat testele definite in folder-ul `test` si le va rula pe rand, afisand la final testele care au esuat, daca exista.

Pentru a va defini propriile teste, creati o noua clasa in folderul `test` care sa extinda clasa `unittest.TestCase` si creati cate o metoda pentru fiecare test. Numele acestor metode trebuie sa inceapa cu `test` pentru a fi recunoscute ca fiind cazuri de testare. Pentru a indica comportamentul testat de fiecare test putem folosi metodele de tipul `self.assert…()` . Unele dintre cele mai frecvent folosite astfel de metode sunt:

- `self.assertTrue(expression_expected_to_be_true)` si `self.assertFalse(expression_expected_to_be_false)`
- `self.assertEqual(expression, expected_value_of_expression)`
- `self.assertIn(expression, list_of_possible_expected_values_of_expression)`

Daca in cadrul unui test vreuna din asertii nu este indeplinita cazul de test este marcat ca esuat.

Scala

Pentru rularea testelor, puteti folosi interfata pusa la dispozitie de IntellIJ. Daca folositi doar command-line, folositi comanda `sbt test` .

Aceasta comanda va rula testele definite in folderul `src/test/scala` si va afisa cu verde testele terminate cu succes si cu rosu testele esuate.

Pentru definirea propriilor teste, creati o noua clasa in folderul `src/test/scala` care sa extinda clasa `munit.FunSuite` , in corpul careia puteti sa adaugati oricate teste sub forma:

```
test("nume test") {
    // instructiuni si asertii
    assert(booleanValue) // -> testul va esua daca booleanValue se evalueaza la fals
}
```

Format arhiva

In radacina proiectului trebuie pus un fisier intitulat `ID.txt` ce va avea pe prima linie a sa ID-ul vostru anonim (ar trebui sa il fi primit pe mail, dar daca din vreun motiv nu il aveti, luati legatura cu asistentul vostru) si pe a doua linie limbajul in care rezolvati tema (`python` sau `scala`)
Exemplu de continut pentru `ID.txt` :

```
9921225
scala
```

sau

```
9246163
python
```

Structura arhivei (Python)

```
.
├── ID.txt
└── src
    ├── DFA.py
    ├── __init__.py
    ├── NFA.py
    ├── Regex.py
    ├── Parser.py
    ... (alte surse pe care le folositi)
```

Structura arhivei (Scala)

```
.
├── build.sbt
├── ID.txt
└── src
    └── main
        └── scala
            ├── Dfa.scala
            ├── Nfa.scala
            ├── Regex.scala
            ... (alte surse pe care le folositi)
```

Pentru niciunul din limbaje nu este necesar sa includeti folder-ul cu teste, dar includerea sa nu va cauza erori.

[1)] Forma prenex, intermediara in proiectul nostru intre cea conventionala si arborele de parsare (AST) construit de programul vostru, nu este standard pentru o astfel de implementare. De altfel, ea nici nu este folosita in mod curent. Insa cum limbajul expresiilor regulate valide este independent de context (din cauza parantezelor), este dificil de implementat parsarea regexurilor fara cunostinte despre gramatici si in special APD-uri. Tocmai de aceea aceasta etapa de parsare, care in mod natural ar fi prima, a fost amanata. Acest lucru a fost posibil introducand forma prenex care nu contine paranteze.