Schelet si checker pentru fiecare limbaj:

- python
- scala

Deadline etapa 1: ~~21.11.2022~~ 23.11.2022 ora 23:00

Proiect

Proiectul consta in implementarea unui lexer in python sau scala.

Ce este un lexer?

Un lexer este un program care imparte un sir de caractere in subsiruri numite *lexeme*, fiecare dintre acestea fiind clasificat ca un *token*, pe baza unei specificatii.

Care este input-ul unui lexer?

Lexer-ul primeste initial o specificatie de forma:

```
TOKEN1 : REGEX1;

TOKEN2 : REGEX2;

TOKEN3 : REGEX3;

...
```

unde fiecare `TOKENi` este un nume dat unui token, iar `REGEXi` este un regex ce descrie lexemele ce pot fi clasificate ca acel token. Puteti imagina aceasta specificatie ca un *fisier de configurare*, care descrie modul in care va functiona lexerul pe diverse fisiere de text.

Inputul efectiv al unui lexer este un text care va fi impartit in lexeme folosind expresii regulate. In cursurile viitoare veti afla mai multe detalii despre cum functioneaza si cum sunt implementate lexerele.

Care este output-ul unui lexer?

Lexer-ul are ca output o lista de forma : `[(lexema1, TOKEN_LEXEMA_1), (lexema2, TOKEN_LEXEMA_2), …]` , unde `TOKEN_LEXEMA_N` este numele token-ului asociat lexemei n, pe baza specificatiei.

Etapa 1

Datorita dificultatii lucrului direct cu regex-uri pentru verificarea apartenentei unui cuvant in limbaj, lexerele reale trec prin cateva etape intermediare inainte de inceperea analizei textului. Aceste etape construiesc un AFD pe baza regex-ului.

Etapa 1 consta in:

1. conversia unei Regex in AFN (folosind Algoritmul Thompson prezentat la curs),
2. conversia unui AFN in AFD (folosind Algoritmul *Subset Construction* prezentat la curs).

Forma de prezentare a unei Regex va fi una mai simpla pentru aceasta etapa, anume **forma Prenex**, explicata mai jos. Parsarea Regex-urilor “standard” (scrise exact ca la curs), va face obiectul etapei urmatoare.

Forma Prenex a expresiilor regulate

Forma Prenex este inspirata din *notatia poloneza* a expresiilor aritmetice (in care, expresii precum `1 + 2 * 3` sunt scrise astfel: `+ 1 * 2 3`. Avantajul acestei notatii (si a formei Prenex in cazul nostru), este ca **parantezele nu mai sunt necesare** pentru a exprima orice expresie. Spre exemplu `(1 + 2) * 3` este scrisa in notatie poloneza astfel: `* + 1 2 3`.

Expresiile regulate in forma Prenex sunt formate din:

- (1) **atomi**
- (2) numele **operatiilor** (`UNION`, `STAR`, `CONCAT`, `PLUS`, `MAYBE`) urmate direct de alte sub-expresii.

Un **atom** poate fi:

- un caracter alfanumeric (e.g. `0` sau `a`)
- un caracter oarecare inclus intre ghilimele simple (e.g. `'a'` sau `';'`)
- unul din cuvintele cheie `eps` (pentru sirul vid) sau `void` (pentru limbajul vid))

Operatiile:

- `PLUS e` (in notatie standard e^+) desemneaza regexul ee^*
- `MAYBE e` (in notatie standard $e?$) desemneaza regexul $e \cup \epsilon$
- iar restul operatiilor au semnificatia lor standard.

Urmatoarele sunt exemple valide de expresii Prenex:

- `UNION a b` , echivalent cu $a \cup b$
- `UNION CONCAT a b STAR c` , echivalent cu $(ab) \cup (c^*)$
- `CONCAT UNION a b UNION c d` , echivalent cu $(a \cup b)(c \cup d)$
- `CONCAT STAR UNION a b UNION b c` , echivalent cu $(a \cup b)^*(b \cup c)$
- `STAR UNION CONCAT a b CONCAT b STAR d` , echivalent cu $((ab) \cup (b(d^*)))^*$
- `CONCAT PLUS c UNION a PLUS b` , echivalent cu $c^+(a \cup (b^+))$
- `UNION ' ' '@'` , accepta limbajul { , @ }
- `UNION eps a`, `MAYBE a` , echivalente cu $a \cup \epsilon$
- `void`

Implementare

Implementarea consta in parsarea expresiei prenex (se recomanda folosirea unei structuri interne arborescente (AST - Abstract Syntax Tree) ca rezultat al parsarii, dar reprezentarea exacta a acestei structuri este la latitudinea voastra) si conversiile Prenex → AFN → AFD, pentru care se vor folosi algoritmii discutati la curs.

Reprezentarea starilor automatelor

Desi cea mai simpla modalitate de a ne referi la o stare este printr-un numar intreg, in anumite componente ale proiectului (si de la aceasta etapa, dar si de la etape viitoare) va fi mult mai convenabil sa lucram cu alte tipuri de etichete pentru stari (de exemplu, seturi de intregi sau tupluri). De aceea, este indicat ca implementarea claselor DFA si NFA trebuie sa fie **generice (polimorfice)** in raport cu tipul de date prin care reprezentam starile.

De asemenea, in mai multe etape ale proiectului va fi necesar sa modificam, in diverse feluri, reprezentarea starilor. Vom realiza acest lucru in cel mai general mod posibil, implementand `map` (putem spune ca AFD-urile si AFN-urile sunt *functori*). Este esential ca implementarea lui `map` sa nu modifice in vreun fel *comportamentul* automatului (adica limbajul acceptat de acesta).

Parsarea expresiilor prenex

Pentru a parsa forma Prenex, 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: `Star(0) | Concat(?,?) | …` , rezultatul reducerii va fi: `Concat(Star(0),?) | …`
- adaugarea expresiilor: vom citi operatorul sau operandul curent, si vom adauga elementele corespunzatoare pe stiva.

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

Testare

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 si AFN-urilor obtinute din cateva expresii in forma prenex, 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
    ... (alte surse pe care le folositi)
```

Structura arhivei (Scala)

```
.
├── build.sbt
├── ID.txt
└── src
    └── main
        └── scala
            ├── Dfa.scala
            ├── Nfa.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.