Schelet si checker pentru fiecare limbaj:

- skel-py-e3.zip
- skel-scala-e3.zip

Deadline etapa 3: 8 ianuarie 2023 ora 23:00 (**deadline hard**)

Proiect

Etapa 3 a 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?

Inputul unul lexer consta in doua componente:

1. o **specificatie**
2. un **text** care va fi analizat lexical, mai precis, impartit in lexeme.

Specificatia are urmatoarea structura:

```
TOKEN1 : REGEX1;

TOKEN2 : REGEX2;

TOKEN3 : REGEX3;

...
```

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

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.

Modalitati de implementare a unui lexer

Exista mai multe modalitati prin care puteti implementa un lexer. Abordarea conventionala (si cea pe care o recomandam) consta in urmatoarele etape:

1. fiecare regex este convertit intr-un AFN, pastrand totodata informatia despre token-ul aferent si pozitia la care acesta apare in spec ().

2. se construieste un AFN unic, introducand o stare initiala si epsilon-tranzitii de la aceasta catre toate starile initiale ale AFN-urilor de mai sus. Astfel, acest AFN va accepta oricare dintre tokenii descrisi in specificatie. Starea finala vizitata va indica token-ul gasit.
3. AFN-ul este convertit la un AFD (care optional poate fi minimizat). In acest automat:
    a. cand vizitam un grup de stari ce contine (AFN-)stari finale, inseamna ca unul sau mai multe token-uri corespunzatoare au fost identificate.
    b. cand vizitam un sink-state (daca acesta exista), inseamna ca subsirul curent nu este descris de nici un token. In acest caz trebuie sa intoarcem cel mai lung prefix acceptat si sa continuam lexarea cuvantului ramas
    c. cand vizitam o stare non-finala si care nu e sink-state, continuam prin trecerea in urmatoarea stare a AFD-ului consumand un caracter din cuvant

Cand isi termina un lexer executia ?

Scopul unui lexer este identificarea **celui mai lung subsir** care satisface un regex din specificatia data. Daca un cel mai lung subsir satisface **doua sau mai multe regex-uri**, va fi raportat primul token aferent, in ordinea in care acestea sunt scrise in specificatie.

Pentru a identifica **cel mai lung subsir** folosind un AFD precum cel descris in sectiunea anterioara, trebuie sa observam faptul ca:

1. vizitarea unui grup de stari ce contine o (AFN-)stare finala, **nu indica** in mod necesar faptul ca am gasit cel mai lung subsir acceptat.
2. daca un grup de stari ce contine o (AFN-)stare finala a fost vizitata **anterior**:
    a. vizitarea unui grup de stari ce nu contine stari finale, **nu indica** in mod necesar faptul ca am gasit cel mai lung subsir (automatul poate accepta in viitor)
    b. vizitarea sink-state-ului AFD-ului (daca acesta exista), indica faptul ca automatul nu va mai accepta in viitor.
    c. daca in AFD nu exista un sink state, atunci analiza lexicala trebuie sa continue pana la epuizarea inputului, pentru a decide asupra celui mai lung subsir.

Odata ce subsirul cel mai lung a fost identificat:

1. AFD-ul va fi *resetat* - adus in starea initiala pentru a relua analiza lexicala.
2. analiza lexicala va continua de la pozitia unde subsirul cel mai lung s-a terminat, iar aceasta poate preceda cu **oricate pozitii**, pozitia curenta unde a ajuns analiza.

Exemplu

Fie specificatia urmatoare:

```
TOKEN1 -> abbc*;
TOKEN2 -> ab+;
TOKEN3 -> a*d;
```

si input-ul `abbd`. Analiza lexicala se va opri la caracterul `d` (AFD-ul descris anterior va ajunge pe acest caracter in sink state). Subsirul `abb` este cel mai lung care satisface atat `TOKEN1` cat si `TOKEN2`, iar `TOKEN1` va fi raportat, intrucat il preceda pe `TOKEN2` In specificatie. Ulterior, lexerul va devansa cu un caracter pozitia curenta in input, si va identifica subsirul `d` ca fiind `TOKEN3`.

Pentru lamuriri ulterioare si mai multe exemple ce includ cel mai lung subsir, revizitati cursul aferent lexerelor.

Structura specificatiei si incarcarea ei

Este recomandat sa folositi functionalitatile implementate la etapele precedente pentru rezolvarea etapei finale

Python

Pentru aceasta etapa va fi nevoie sa implementati clasa `Lexer` cu 2 metode obligatorii:

- constructorul care primeste ca parametru configuratia lexerului
- metoda `lex` care va primi un cuvand ca `str` si va intoarce rezultatul lexarii lui sub forma `List[Tuple[str, str]] | str`. Metoda `lex` va intoarce o lista de tupluri `(token, lexem_cuvant)` in cazul in care lexarea reuseste si un string cu un mesaj de eroare in caz contrar. (*Mai multe despre cazurile in care un lexer poate esua mai jos*)

Specificatia va fi incarcata in teste sub forma unui dictionar `TOKEN → REGEX`

Scala

La aceasta etapa va trebui sa implementati clasa `Lexer` care va primi ca parametru o specificatie sub forma de `String`. Specificatia va fi data sub forma

```
TOKEN1: REGEX1;
TOKEN2: REGEX2;
TOKEN3: REGEX3;
```

Metoda `lex` va imparti cuvantul dat ca parametru intr-o lista de lexeme sub forma `(LEXEM, TOKEN)`. Ea va intoarce un `Either[String,List[(String,String)]]` deoarece in caz de succes vrem sa intoarcem `Right(LISTA_LEXEME)` si in caz de eroare `Left(ERROR_MESSAGE)` (*Mai multe despre cazurile in care un lexer poate esua mai jos*)

Parser pentru un limbaj de programare

Ultimul test din fisierul de teste va testa lexerul scris de voi pe sample-ul de cod real, folosind un limbaj super simplificat.

Pentru a rezolva aceasta parte va trebui sa scrieti voi configuratia pentru toate tokenurile prezente in limbaj. Un scehelet pentru aceasta configuratie se afla in fisierul `configuration.json` in python si `configuration` in scala. Tokenurile prezente in schelet sunt toate tokenurile care pot sa apara in acest limbaj, singurul lucru care va ramane voua fiind scrierea unor regexuri care accepta aceste tokenuri.

Tokenurile sunt:

```
  "BEGIN":
  "END":
  "EQUAL":
  "ASSIGN":
  "PLUS":
  "MINUS":
  "MULTIPLY":
  "GREATER":
  "WHILE":
  "DO":
  "OD":
  "IF":
  "THEN":
  "ELSE":
  "FI":
  "RETURN":
  "OPEN_PARANTHESIS":
  "CLOSE_PARANTHESIS":
  "NUMBER":
  "VARIABLE":
```

Fiecare token va trebui sa accepte o componenta din cod, spre exemplu tokenul `BEGIN` va trebui sa accepte cuvantul `begin` care se foloseste la inceputul unui scope nou.

Codul furnizat are prezente si whitespace uri (space-uri, tab-uri si new-line uri). Nu avem un token special pentru aceste whitespaces, si pentru a simplifica complexitatea output-ului va trebui sa considerati pentru fiecare token un numar arbitrar de whitespaces in jurul lui, important fiind doar ca el sa contina tokenul pe care incearca sa il accepte.

Testerul va testa doar ca ordinea tokenurilor sa fie corecta, nu si ce lexeme vor accepta aceste tokenuri

Exemplu: pentru codul

```
begin
a = 1
end
```

va trebui sa afisam tokenurile in ordinea

```
"BEGIN", "VARIABLE", "ASSIGN", "NUMBER", "END"
```

Python

Configuratia va fi scrisa respectand formatul `json` al fisierului, si anume veti pune regexul fiecarui token intre ghilimele pe o linie separata

Exemplu:

```
{
    "BEGIN": "s*me|r*g*x",
    "END": "oth*r|r+g+x",
    ...
}
```

Scala

Configuratia va fi scrisa pe aceeasi linie cu tokenul lasand un spatiu liber si un `;` la final

Exemplu:

```
"BEGIN": s*me|r*g*x;
"END": oth*r|r+g+x;
...
```

In scala eroare va fi intoarsa sub forma unui `Either[String]` in formatul `Left(message)`

Erori de lexare

Erorile de lexare sunt in general cauzate o configuratie gresita / incompleta sau de un cuvant invalid. Informatiile care trebuie transmise in acest caz trebuie sa ajute programatorul sa isi dea seama unde un cod s-a intamplat eroare si care este tipul erorii. Din acest motiv vom afisa linia si coloana unde lexarea a esuat si tipul erorii.

- Unordered List ItemEroare cauza de un caracter invalid in cuvant

Aceasta eroare va aparea daca lexarea s-a oprit fara a accepta nici-un cuvant in prealabil. Aceasta este echivalenta cu ajungerea in starea `SINK_STATE` a lexerului fara a trece in prealabil printr-o stare finala. In acest caz vom afisa un mesaj de eroare in formatul

```
No viable alternative at character ..., line ...
```

In primul loc liber vom pune indexul caracterului unde s-a oprit lexarea (am ajuns in `SINK_STATE`) indexat de la 0, iar in an doilea spatiu liber vom pune linia unde s-a intamplat asta (indexata de la 0).

- Unordered List ItemEroare cauza de un cuvant incomplet

Aceasta eroare va aparea daca lexarea a ajuns la finalul cuvantului fara a accepta in prealabil un lexem. In aceasta stare lexerul nu a ajuns in sink state, insa nici intr-o stare finala. In acest caz vom afisa un mesaj de eroare in formatul:

```
No viable alternative at character EOF, line ...
```

Ca un mic rezumat: prima eroare apare atunci cand caracterul la care am ajuns este invalid si nu avem cum sa acceptam, iar a doua apare atunci cand lexerul ar mai accepta, insa cuvantul este incomplet si nu mai are ce.

Python

In python eroare va fi intoarsa sub forma unui `string` de functa `lex`

Scala

In scala eroare va fi intoarsa sub forma unui `Either[String]` in formatul `Left(message)`

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
    ├── Lex.py
    ... (alte surse pe care le folositi)
```

Structura arhivei (Scala)

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