Tema 2 PP 2024

DEADLINE 5 MAI 23:59

- Temele trebuie submise pe curs.upb.ro, in assignment-ul numit `Tema 2 Scala`.
- Temele ce nu sunt acceptate de validatorul de arhive **NU** vor fi puncate.
- Vom folosi vesiunea Scala **3.3.1** si **JDK 21**.

Folosiți un stil de programare funcțional. NU se vor accepta:

- **Efecte laterale** (de exemplu modificarea parametrilor dați ca input la funcție)
- **var** (**val** este ok!)

Scheletul se poate găsi la:skel2.zip

Scopul Temei

In cadrul acestei teme veti implementa un Query Language inspirat de SQL, ce utilizeaza o baza de date implementata de voi. Va vom ghida in realizarea operatiunilor atat pe tabele individuale, cat si pe interactiuni intre mai multe tabele, pe care apoi le veti putea combina. Mai mult, vom adopta folosirea sintaxei de `extension` si `implicits` pentru a simplifica apelurile de functii, oferind in acest mod o sintaxa pentru interogari care este usor de inteles si eleganta.

Reprezentarea Tabelelor

Considerati exemplul de mai jos.

Nume	Prenume	Varsta
Popescu	Ion	30
Ionescu	Maria	25

Acest tabel poate fi reprezentat ca:

```
type Row = Map[String, String] // nume_coloana - valoare
type Tabular = List[Row]
```

Funcții utile pentru lucrul cu Map

```
let map = Map(1 -> 2, 3 -> 4): Map[Int, Int]
```

- Adauga o noua pereche cheie-valoare

```
map + (5 -> 6)  // Map(1 -> 2, 3 -> 4, 5 -> 6)
map + (3 -> 5)  // Map(1 -> 2, 3 -> 5)  -- if key exists, it updates the value
```

- Eliminarea unei perechi cheie-valoare

```
map - (3 -> 4)  // Map(1 -> 2)
```

- Accesarea valorii asociate unei chei

```
map get 1  // return 2
map get 3  // return 4
map getOrElse (1, 0)  // return 2
map getOrElse (5, 0)  // return 0 - daca cheia nu exista, returneaza valoarea default
map contains 1  // True
map contains 5  // False
```

- Functii de orin superior

```
map mapValues (x => x + 5)  // Map(1 -> 7, 2 -> 9)
map filterKeys (x => x <= 2)  // Map(1 -> 2)
```

- Combinarea a doua map-uri

```
val map1: Map[Int, Int] = Map(1 -> 2, 3 -> 4)
val map2: Map[Int, Int] = Map(5 -> 6, 7 -> 8)
map ++ map2  // Map(1 -> 2, 3 -> 4, 5 -> 6, 7 -> 8)
```

Clasa Table

Vom defini un tabel ca o clasa care are ca atribute numele tabelei `tableName` si datele `tableData`.

```
case class Table (tableName: String, tableData: Tabular) {
  def header: List[String] = ???
  def data: Tabular = ???
  def name: String = ???
```

```
}
```

1.1. Definiti metoda `toString` care returneaza tabelul in forma CSV.

```
override def toString: String = ???
```

1.2 Definiti operatia de inserare a unei linii in tabel.

```
def insert(row: Row): Table = ???
```

1.3 Definiti operatia de stergere a tuturor liniilor exact egale cu cea primita ca parametru.

```
def delete(row: Row): Table = ???
```

1.4. Definiti operatia de sortare a liniilor din tabel dupa o anumita coloana.

```
def sort(column: String): Table = ???
```

1.5. Definiti functia select care primeste o lista de stringuri si returneaza un nou obiect de tip Table ce contine doar coloanele specificate.

```
def select(columns: List[String]): Table = ???
```

1.6. Definiti functia apply intr-un **companion object** al clasei `Table`. Functia trebuie sa parseze un sir de caractere si sa returneze un tabel cu numele dat.

```
def apply(name: String, s: String): Table = ???
```

Filtre peste Tabele

```
<filter> ::=
   <filter> && <filter> |
   <filter> || <filter> |
   <filter> == <filter> |
   !<filter> |
   any [ <filter> ] |
   all [ <filter> ] |
   operation [ <filter> ]
```

2.1. Vom defini operatia de filtrare a datelor dintr-o tabela sub forma unui TDA. Acest lucru ne permite sa definim operatii de filtrare complexe, compuse din mai multe conditii. Acest TDA are urmtorii constructori:

1. Field - reprezinta o conditie de filtrare pe un camp al tabelei. Aceasta conditie este satisfacuta daca valoarea de pe coloana specificata respecta predicatul.
2. Compound - reprezinta o conditie de filtrare compusa din mai multe conditii. Aceasta conditie este satisfacuta daca toate conditiile din lista conditions sunt satisfacute.
3. Not - reprezinta negarea unei conditii de filtrare.
4. And* - reprezinta conjunctia a doua conditii de filtrare.
5. Or* - reprezinta disjunctia a doua conditii de filtrare.
6. Equal* - reprezinta o conditie de egalitate intre doua conditii de filtrare.
7. Any - reprezinta o conditie de filtrare care este satisfacuta daca cel putin una dintre conditiile din lista este satisfacuta.
8. All - reprezinta o conditie de filtrare care este satisfacuta daca toate conditiile din lista sunt satisfacute.

```
trait FilterCond { def eval(r: Row): Option[Boolean] }


case class Field(colName: String, predicate: String => Boolean) extends FilterCond {
  override def eval(r: Row): Option[Boolean] = ???
}


case class Compound(op: (Boolean, Boolean) => Boolean, conditions: List[FilterCond])
extends FilterCond {
  override def eval(r: Row): Option[Boolean] = ???
}


case class Not(f: FilterCond) extends FilterCond {
  override def eval(r: Row): Option[Boolean] = ???
}


def And(f1: FilterCond, f2: FilterCond): FilterCond = ???
def Or(f1: FilterCond, f2: FilterCond): FilterCond = ???
def Equal(f1: FilterCond, f2: FilterCond): FilterCond = ???


case class Any(fs: List[FilterCond]) extends FilterCond {
  override def eval(r: Row): Option[Boolean] = ???
}


case class All(fs: List[FilterCond]) extends FilterCond {
  override def eval(r: Row): Option[Boolean] = ???
```

```
}
```

2.2. Pentru a simplifica definirea conditiilor de filtrare, vom defini cateva operatori care sa ne permita sa scriem cod mai concis. Vom folosi urmatorii operatori ce extind clasa FilterCond:

1. === - pentru a verifica egalitatea a doua conditii de filtrare.
2. && - pentru a face conjunctia a doua conditii de filtrare.
3. || - pentru a face disjunctia a doua conditii de filtrare.
4. !! - pentru a nega o conditie de filtrare.

```
extension (f: FilterCond) {
  def ==(other: FilterCond) = ???
  def &&(other: FilterCond) = ???
  def ||(other: FilterCond) = ???
  def !! = ??
  // Puteti sa adaugati mai multi operatori :)
}
```

2.3. In plus vom abstractiza instantierea unui obiect Field, astfel incat sa putem folosi un tuplu de forma (String, String ⇒ Boolean) pentru a crea un obiect Field.

```
implicit def tuple2Field(t: (String, String => Boolean)): Field = ???
```

2.4. Definiti operatia de filtrare a liniilor din tabel care respecta o anumita conditie.

```
def filter(f: FilterCond): Table = ???
```

2.5. Definiti operatia de update a unei linii din tabel. Funcția primeste ca input o conditie care dicteaza liniile ce vor fi modificate. Valorile schimbate se găsesc intr-un Map[nume_coloana, valoare_noua].

```
def update(f: FilterCond, updates: Map[String, String]): Table = ???
```

Operatii cu una sau mai multe Tabele

O baza de date contine mai multe tabele, pe care putem aplica o serie de operatii:

1. create - creaza o noua tabela cu nume unic si o lista de coloane
2. drop - sterge o tabela existenta
3. selectTables - extrage din lista de tabele existente un subset de tabele
4. join - combina doua tabele pe baza unei chei comune

Pentru a gestiona operatii cu una sau mai multe tabele, vom folosi clasa:

```
case class Database(tables: List[Table]) {
```

```
  override def toString: String = ???
}
```

3.1.. Implementati functia create, care primeste numele unei tabele si creeaza o noua tabela doar daca numele tabelei nu exista deja in baza de date. Daca numele tabelei exista, functia va intoarce baza de date nemodificata.

```
def create(tableName: String): Database = ???
```

3.2. Implementati functia drop, care primeste numele unei tabele si sterge tabela respectiva din baza de date. Daca numele tabelei nu exista in baza de date, functia va intoarce baza de date nemodificata.

```
def drop(tableName: String): Database = ???
```

3.3. Implementati functia selectTables care primeste o lista de nume de tabele si extrage din baza de date doar acele tabele. Daca unul dintre numele de tabele nu exista in baza de date, functia va intoarce None.

```
def selectTables(tableNames: List[String]): Option[Database] = ???
```

3.4 Mai adăugati ceva (cautati voi functia de implementat 😀) la Table astfel incat sa putem accesa Rows din tabel folosind un index. Faceți același lucru si pentru Database ca sa putem accesa tabelele sale folosind index direct din numele unei instante.

```
val tabel = new Table("People", List(
  Map("id" -> "1", "name" -> "John", "age" -> "23", "CNP" -> "1234567890123"),
  Map("id" -> "2", "name" -> "Jane", "age" -> "25", "CNP" -> "1234567890124"),
  Map("id" -> "3", "name" -> "Jack", "age" -> "27", "CNP" -> "1234567890125"),
  Map("id" -> "4", "name" -> "Jill", "age" -> "29", "CNP" -> "1234567890126"),
))(1) // index aici
val dbPeople = Database(List(tabel))
val tabel2 = dbPeople(0) // index aici
```

3.5. Implementati functia join, care primeste doua tabele si o coloana specifica pentru fiecare tabel. Aceasta functie va combina tabelele folosind coloanele indicate, rezultand intr-un nou tabel.

- Cand valorile din coloanele folosite pentru combinatie sunt identice, se va retine o singura valoare din acele coloane.
- Daca valorile difera, ele vor fi unite intr-un singur camp, separandu-le prin semnul “;”, urmand ordinea in care tabelele sunt enumerate in apelul functiei.
- Se considera ca valorile de tip sir de caractere gol (“”) sunt echivalente cu NULL, adica acestea nu vor fi incluse daca exista o valoare specifica intr-o alta tabela.
- In situatiile in care o linie este prezenta in tabelul A dar nu are corespondent in tabelul B, se vor completa campurile corespunzatoare din tabelul B cu sirul vid “”.

- Similar, daca o linieeste prezenta in tabelul B dar nu are corespondent in tabelul A, se vor completa campurile corespunzatoare din tabelul A cu sirul vid “”.
- Numele coloanei utilizate pentru join in tabelul final va fi preluat din primul tabel.
- Se va intoarce eroare cand unul din tabele nu exista. Daca un tabel este gol, se va intoarce celălalt tabel.
- Liniile din rezultat sunt in ordinea: linii ce au intrări in ambele tabele, linii doar in prima tabela, linii doar in a2a tabela.

Exemplu

Exemplu:
Tabelul A

id	name	age
1	Ana	20
2	Ion	30
4	Maria	

Tabelul B

id	city	job	age
1	Cluj	IT	20
2	Iasi	HR	
3	Buc	MKT	40
4	Buc		

join(“A”, “id”, “B”, “id”)

id	name	age	city	job
1	Ana	20	Cluj	IT

id	name	age	city	job
2	Ion	30	Iasi	HR
3		40	Buc	MKT
4	Maria		Buc	

```
def join(table1: String, c1: String, table2: String, c2: String): Option[Table] = ???
```

Query Language

Vom dezvolta un limbaj de interogare, care va servi ca API pentru o gama variata de transformari de tabele, anterior implementate sub forma de functii. Acest limbaj de query va permite secvente sau combinatii ale acestor transformari.

In cadrul implementarii limbajului de interogare, ne vom concentra pe includerea functionalitatilor asemanatoare cu cele din SQL, precum si pe gestionarea erorilor. Limbajul va permite doua categorii principale de operatii:

- operatii pe toată baza de date
- operatii pe o singura tabela

Pentru tratarea erorilor, vom utiliza TDA-ul `Option`, unde `Some(_)` indica un rezultat valid al unei operatii, in timp ce `None` semnaleaza o eroare. In cazul in care un query genereaza o eroare, aceasta se va propaga daca rezultatul este necesar in executarea unui alt query.

4.1. Vom defini operatiile ce se pot realiza pe o baza de date folosind TDA-ul `PP_SQL_DB`.
Funcția `eval` trebuie sa apeleze metodele corespunzatoare definite in `Database`.

```
trait PP_SQL_DB{
  def eval: Option[Database]
}

case class CreateTable(database: Database, tableName: String) extends PP_SQL_DB{
  def eval: Option[Database] = ???
}

case class DropTable(database: Database, tableName: String) extends PP_SQL_DB{
  def eval: Option[Database] = ???
```

```
}


case class SelectTables(database: Database, tableNames: List[String]) extends
PP_SQL_DB{

  def eval: Option[Database] = ???

}


case class JoinTables(database: Database, table1: String, column1: String, table2:
String, column2: String) extends PP_SQL_DB{

  def eval: Option[Database] = ??? // conventie: intoarce un Database ce conține o
singura tabela

}
```

4.2. Vom defini operatiile ce se pot realiza pe o tabela folosind TDA-ul `PP_SQL_Table`.
Funcția `eval` trebuie sa apeleze metodele corespunzatoare definite in `Table`.

```
trait PP_SQL_Table{

  def eval: Option[Table]

}


case class InsertRow(table:Table, values: Tabular) extends PP_SQL_Table{

  def eval: Option[Table] = ???

}


case class UpdateRow(table: Table, condition: FilterCond, updates: Map[String,
String]) extends PP_SQL_Table{

  def eval: Option[Table] = ???

}


case class SortTable(table: Table, column: String) extends PP_SQL_Table{

  def eval: Option[Table] = ???

}


case class DeleteRow(table: Table, row: Row) extends PP_SQL_Table{

  def eval: Option[Table] = ???

}
```

```
case class FilterRows(table: Table, condition: FilterCond) extends PP_SQL_Table{
  def eval: Option[Table] = ???
}

case class SelectColumns(table: Table, columns: List[String]) extends PP_SQL_Table{
  def eval: Option[Table] = ???
}
```

Nota: Am vrea sa avem o sintaxa mai usor de citit pentru aces Query Language. De aceea, vom defini `implicits` pentru fiecare din operațiile `eval` ale acestor 2 TDA-uri. Forma unui query, fie ca este pe toată baza de date, fie ca este pe o singura tabela are forma unui tuplu de tipul:

```
(tabel, "OPERATIE", ...parametri...)
(   db, "OPERATIE", ...parametri...)
```

unde operatia este un string:

- CreateTable - “CREATE”
- DropTable - “DROP”
- SelectTables - “SELECT”
- JoinTables - “JOIN”
- InsertRow - “INSERT”
- UpdateRow - “UPDATE”
- SortTable - “SORT”
- DeleteRow - “DELETE”
- FilterRows - “FILTER”
- SelectColumns - “EXTRACT”

4.3. Implementati functii de conversie implicite intre tuplurile descrise mai sus si query-ul descris de acestea.

Nota: Erorile ce apar in cardul primului element din cuplu vor fi propagate la rezultat, pentru a putea ulterior combina query-uri.

```
implicit def PP_SQL_DB_Create_Drop(t: (Option[Database], String, String)):
Option[PP_SQL_DB] = ??? // 2 operatii combinate pentru ca au aceeasi structura a
parametrilor


implicit def PP_SQL_DB_Select(t: (Option[Database], String, List[String])):
Option[PP_SQL_DB] = ???


implicit def PP_SQL_DB_Join(t: (Option[Database], String, String, String, String,
String)): Option[PP_SQL_DB] = ???
```

```
implicit def PP_SQL_Table_Insert(t: (Option[Table], String, Tabular)):
Option[PP_SQL_Table] = ???


implicit def PP_SQL_Table_Sort(t: (Option[Table], String, String)):
Option[PP_SQL_Table] = ???


implicit def PP_SQL_Table_Update(t: (Option[Table], String, FilterCond, Map[String,
String])): Option[PP_SQL_Table] = ???


implicit def PP_SQL_Table_Delete(t: (Option[Table], String, Row)):
Option[PP_SQL_Table] = ???


implicit def PP_SQL_Table_Filter(t: (Option[Table], String, FilterCond)):
Option[PP_SQL_Table] = ???


implicit def PP_SQL_Table_Select(t: (Option[Table], String, List[String])):
Option[PP_SQL_Table] = ???
```

Query

Acum, dupa implementarea tuturor operatiunilor, veti observa ca, in momentul in care incercati sa compuneti query-uri folosind functiile mentionate anterior, este necesar sa:

- extrageti valori din `Option`
- invocati manual functia “eval” pentru a obtine rezultatul unui subquery

5.1. Pentru ca nu ne place sa facem adnotari explicite in Query Language, vom defini functiile de conversie de mai jos:

```
def queryT(p: Option[PP_SQL_Table]): Option[Table] = ???

def queryDB(p: Option[PP_SQL_DB]): Option[Database] = ???
```

Ne vom opri aici cu `implicits` ca deja parca ca nu mai scriem cod Scala :) Aceasta functionalitate este foarte utila, dar trebuie folosita limitat, altfel ajungem sa nu mai putem urmari ce se intampla in cod. In continuare doar vom scrie cateva query-uri, aplicand tot ce am implementat pana acum.

Pentru funcțiile de mai jos, scrieti query-ul ca un ONE-LINER.

5.2. Scrieti o functie care utilizeaza limbajul de interogare implementat pentru a elimina dintr-o tabela liniile unde valoarea `Jack` apare in coloana `name`.

```
def killJackSparrow(t: Table): Option[Table] = ???
```

5.3. Scrieti o functie care insereaza o tabela numita “Inserted Fellas” in baza de date, apoi selecteaza aceasta tabela si insereaza urmatoarele persoane:

1. numele Ana, varsta 93, CNP 455550555
2. numele Diana, varsta 33, CNP 255532142
3. numele Tatiana, varsta 55, CNP 655532132
4. numele Rosmaria, varsta 12, CNP 855532172

Apoi sortati tabelul dupa varsta.

```
def insertLinesThenSort(db: Database): Option[Table] = ???
```

5.4. Combina tabelele People si Hobbies pe baza coloanelor name. Filtreaza rezultatul astfel incat sa ramana doar young adults cu varsta sub 25 ani, al caror nume incepe cu 'J' si care au un hobby. In final se extrag doar coloanele name si hobby.

```
def youngAdultHobbiesJ(db: Database): Option[Table] = ???
```

Testare

`Scalatest` este o biblioteca de testare pentru Scala care suporta mai multe stiluri de scriere a testelor, inclusiv testarea traditionala unitara. `Scalactic` este o biblioteca destinata sa faciliteze scrierea de cod mai clar si mai intretinabil in Scala, utilizata in combinatie cu Scalatest pentru a imbunatati claritatea si precizia testelor.

Pentru a rula testele utilizand aceasta configuratie, puteti folosi comanda de mai jos in terminal, de la radacina proiectului. Acest lucru va compila si executa toate testele definite in proiect care depind de Scalatest si Scalactic pentru a verifica corectitudinea codului.

```
sbt test
```

Submisie arhiva

Veti incarca pe moodle o arhiva ce contine, in radacina acesteia, folderul `src` al proiectului vostru, fisierul `build.sbt` si un fisier text, intitulat `ID.txt` ce contine o singura linie, si anume id-ul vostru anonim (pe care il puteti gasi pe moodle la assignment-ul `tokenID`).

Exemplu structura arhiva:

```
archive.zip
|-src/
| |-main/
| | |-scala/
| | | | - ... <fisierele cu sursa scala>
```

```
|-build.sbt
|-ID.txt
```

Puncatje

Pentru ca avem cam multe exercitii de implementat, găsiți mai jos un tabel cu punctajele grupate:

Parte a temei	**Functionalitate**	**Punctaj**
Table	toString	0.5
Table	insert	1.5
Table	delete	1.5
Table	sort	1.5
Table	select	1.5
Table	apply	1.5
Table	filter	4.5
Table	update	4.5
TABLE	**TOTAL**	**17**
Filter	Field	4
Filter	Compound	3

Parte a temei	Functionalitate	Punctaj
Filter	Not	2
Filter	And	2
Filter	Or	2
Filter	Equal	2
Filter	Any	2
Filter	All	2
Filter	implicit equal	0.5
Filter	implicit or	0.5
Filter	implicit and	0.5
Filter	implicit not	0.5
Filter	implicit touple2Field	1
FILTER	**TOTAL**	**22**
Database	create	1.5
Database	drop	1.5

Parte a temei	Functionalitate	Punctaj
Database	selectTables	1.5
Database	indexing	1.5
Database	join	10
DATABASE	**TOTAL**	**16**
QueryLanguage	CreateTable	2
QueryLanguage	DropTable	2
QueryLanguage	Create & Drop implicit	1
QueryLanguage	SelectTables	2
QueryLanguage	Select implicit	1
QueryLanguage	JoinTables	2
QueryLanguage	Join implicit	1
QueryLanguage	InsertRow	2
QueryLanguage	Insert implicit	1
QueryLanguage	UpdateRow	2

Parte a temei	Functionalitate	Punctaj
QueryLanguage	Update implicit	1
QueryLanguage	SortRow	2
QueryLanguage	Sort implicit	1
QueryLanguage	DeleteRow	2
QueryLanguage	Delete implicit	1
QueryLanguage	FilterRows	2
QueryLanguage	Filter implicit	1
QueryLanguage	SelectColumns	2
QueryLanguage	Extract implicit	1
QUERY LANGUAGE	**TOTAL**	**30**
Queries	queryT	0
Queries	queryB	0
Queries	killJackSparrow	5
Queries	insertLinesThenSort	5

Parte a temei	Functionalitate	Punctaj
Queries	youngAdultHobbies	5
QUERIES	**TOTAL**	**15**
`TEMA 2`	`TOTAL`	`100`