7. homework assignment; JAVA, Academic year 2016/2017; FER

Napravite prazan Maven projekt, kao u 1. zadaći: u Eclipsovom workspace direktoriju napravite direktorij `hw07-0000000000` (zamijenite nule Vašim JMBAG-om) te u njemu oformite Mavenov projekt `hr.fer.zemris.java.jmbag0000000000:hw07-0000000000` (zamijenite nule Vašim JMBAG-om) i dodajte ovisnost prema `junit:junit:4.12`. Importajte projekt u Eclipse. Sada možete nastaviti s rješavanjem zadataka.

Skrećem pažnju: rješenje ove zadaće koristit ćete kao osnovu za sljedeću domaću zadaću.

Razvijamo sustav koji će nam omogućiti rad s Booleovim funkcijama. U okviru ove domaće zadaće napravit ćete podršku za nekoliko osnovnih operacija.

Pa krenimo redom.

Problem 1.

Napravite paket `hr.fer.zemris.bf.lexer`. U njega smjestite razrede `Lexer`, `LexerException` (izveden iz `RuntimeException`) i `Token` te enum `TokenType` (`EOF`, `VARIABLE`, `CONSTANT`, `OPERATOR`, `OPEN_BRACKET`, `CLOSED_BRACKET`). Razred `Lexer` nudi sljedeće javno sučelje (pojam "javno sučelje" ovdje ne predstavlja doslovno sučelje u smislu ključne riječi `interface`, već popis svega javnoga što je vidljivo – u nastavku je to jedan javni konstruktor i jedna javna članska metoda):

```
public Lexer(String expression);
public Token nextToken();
```

Razred `Token` nudi sljedeće javno sučelje:

```
public Token(TokenType tokenType, Object tokenValue);
public TokenType getTokenType();
public Object getTokenValue();
public String toString();
```

Evo primjera:

```
try {
	Lexer lexer = new Lexer("(false or b) xor not (a or c)");
	Token token = null;
	do {
		token = lexer.nextToken();
		System.out.println(token);
	} while(token.getTokenType()!=TokenType.EOF);
} catch(LexerException ex) {
	System.out.println("Iznimka: " + ex.getClass()+" - " + ex.getMessage());
}
```

Lexer obrađuje ulaz koji predstavlja Booleov izraz. Booleov izraz se sastoji od konstanti, varijabli, operatora te otvorenih i zatvorenih zagrada. Prilikom izrade lexera, pridržavajte se sljedećih pravila.

Identifikator je svaki slijed znakova koji počinje slovom (`Character#isLetter`) nakon čega slijedi nula ili više slova ili brojeva (`Character#isLetterOrDigit`) ili znakova podvlake.

Numerički niz je svaki niz koji počinje znamenkom i nakon toga slijedi nula ili više znamenaka.
Tokeni koje lexer generira razrješavaju se prema prioritetima (od vrha prema dnu) kako je napisano u nastavku.

Praznine (razmaknice, tabovi, enteri) se preskaču.

Ako je lexer pronašao *identifikator*:

1. ako je to `"and"`, `"xor"`, `"or"`, `"not"` ili bilo koja kombinacija velikih i malih slova tih riječi, lexer vraća token tipa `OPERATOR` čija je vrijednost string naziva operatora (uvijek malim slovima);
2. ako je to `"true"` ili `"false"` ili bilo koja kombinacija velikih i malih slova tih riječi, lexer vraća token tipa `CONSTANT` čija je vrijednost odgovarajući `Boolean` objekt;
3. inače se vraća token tipa `VARIABLE` čija je vrijednost `String` čija su sva slova velika.

Ako je lexer pronašao *numerički niz*:

1. ako je to `0` ili `1`, lexer vraća token tipa `CONSTANT` čija je vrijednost odgovarajući `Boolean` objekt (`true` za `1`, `false` za `0`);
2. inače se baca iznimka.

Ako je lexer pronašao `'('` ili `')'`, vraća se token tipa `OPEN_BRACKET` odnosno `CLOSED_BRACKET` čija je vrijednost `Character` objekt koji predstavlja pronađeni simbol.

Ako je lexer pronašao `'*'`, `'+'`, `'!'`, `':+:'`, vraća se token tipa `OPERATOR` čija je vrijednost string naziva operatora malim slovima (redom: `"and"`, `"or"`, `"not"`, `"xor"`).

Ako ništa od prethodnoga ne pokriva pronađeno, lexer baca iznimku.

U okviru ovog zadatka pripremio sam Vam i nekoliko demonstracijskih programa. ZIP arhiva uploadana je kao privitak. Priložene programe trebate ubaciti u paket `demo` redosljedom kojim ih ovdje navedemo. Nemojte ih u projekt dodavati prije toga jer ćete imati niz pogrešaka zbog referenciranja nepostojećih razreda odnosno metoda. Za svaki program datoteka istog imena ali ekstenzije txt sadrži ispis programa kakav trebate dobiti kada pokrenete taj program.

Ubacite u projekt demonstracijski program `Izrazi1.java`. Napišite lexer i zadane razrede, te osigurajte da Vam ispitni program daje izlaz kakav je prikazan u `Izrazi1.txt`.

Problem 2.

Napravite paket `hr.fer.zemris.bf.parser`. U njega smjestite razrede `Parser`, i `ParserException` (izveden iz `RuntimeException`). Javno sučelje razreda `Parser` treba biti:

```
public Parser(String expression);
public Node getExpression();
```

Parser izvedite kao parser rekurzivnog spusta, čija je gramatika (S je početni nezavršni simbol):

```
S  -> E1
E1 -> E2 (OR E2)*
E2 -> E3 (XOR E3)*
E3 -> E4 (AND E4)*
E4 -> NOT E4 | E5
E5 -> VAR          | KONST     | '(' E1 ')'
```

U prethodnoj gramatici, crvenim su označeni tokeni. Pravila za `E1`, `E2` i `E3` napisana su na način da dozvoljavaju da se operator dan u pravilu pojavi uzastopno nula, jednom ili više puta. Primjerice, naleti li parser negdje na niz `a or b or c or d`, to treba "shvatiti" kao primjenu operatora `OR` nad četiri djeteta: listom (`a`, `b`, `c`, `d`).

Zadaća parsera je izgradnja stabla koje predstavlja parsirani izraz.

Napravite paket `hr.fer.zemris.bf.model`. U njega ćemo smjestiti sučelja i razrede koji modeliraju ovo stablo i operacije nad njime. Svaki čvor stabla modeliran je sučeljem `Node`:

```
public interface Node {
      void accept(NodeVisitor visitor);
}
```

Konstante su modelirane čvorom tipa `ConstantNode`. Konstruktor prima vrijednost konstante i pamti je.

```
public ConstantNode(boolean value) {...}
public void accept(NodeVisitor visitor) {...}
public boolean getValue() {...}
```

Varijable su modelirane čvorom tipa `VariableNode`. Konstruktor prima ime varijable i pamti je.

```
public VariableNode(String name) {...}
public void accept(NodeVisitor visitor) {...}
public String getName() {...}
```

Unarne operacije (poput komplementiranja) modelirane su čvorom tipa `UnaryOperatorNode`. Konstruktor prima naziv operatora, referencu na čvor koji predstavlja operand nad kojim treba djelovati te strategiju koja implementira djelovanje samog operatora.

```
public UnaryOperatorNode(String name, Node child, UnaryOperator<Boolean> operator)
{...}
public void accept(NodeVisitor visitor) {...}
public String getName() {...}
public Node getChild() {...}
public UnaryOperator<Boolean> getOperator() {...}
```

Binarne operacije (poput i, ili, ...) modelirane su čvorom tipa `BinaryOperatorNode`. Konstruktor prima naziv operatora, listu referenci na proizvoljan broj čvorova koji predstavljaju operande nad kojim treba djelovati (minimalno ih mora biti 2 ali ih može biti i više) te strategiju koja implementira djelovanje samog binarnog operatora.

```
public BinaryOperatorNode(String name, List<Node> children, BinaryOperator<Boolean>
operator) {...}
public void accept(NodeVisitor visitor) {...}
public String getName() {...}
public List<Node> getChildren() {...}
public BinaryOperator<Boolean> getOperator() {...}
```

Operacije koje ćemo izvoditi nad stablom riješit ćemo primjenom oblikovnog obrasca *Posjetitelj* (engl. *Visitor*). Primjer ovog oblikovnog obrasca opisan je u knjizi u poglavlju *Studija slučaja: jezik Vlang*, a i Internet obiluje informacijama. Ako niste sigurni jeste li dobro shvatili ideju ovog obrasca, slobodno me potražite pa pitajte. Ovaj oblikovni obrazac koristit ćete u još nekim domaćim zadaćama.

Posjetitelji su kod nas modelirani sučeljem `NodeVisitor`.

```
void visit(ConstantNode node);
void visit(VariableNode node);
void visit(UnaryOperatorNode node);
void visit(BinaryOperatorNode node);
```

Svaki čvor stoga navodi metodu `accept` koja prima referencu nad posjetiteljem, i nad njim poziva metodu koja odgovara upravo tipu samog čvora.

Napišite prethodno definirane razrede i sučelja modela, i potom napišite parser.

Najjednostavniji način da provjerite jeste li dobro implementirali parser jest da pokušate ispisati stablo koje je nastalo. U tu svrhu napisat ćete Vašeg prvog posjetitelja: razred `ExpressionTreePrinter`. Sve posjetitelje smjestit ćemo u paket `hr.fer.zemris.bf.utils`. Napravite taj paket i u njemu razred `ExpressionTreePrinter` koji implementira sučelje `NodeVisitor`. Zadaća ovog posjetitelja jest da na zaslon ispisuje nastalo stablo uz vođenje računa o indentaciji (svaki puta kada uđe u neki operator, indentaciju treba povećati za 2). Evo primjera uporabe i generiranog ispisa.

```
Parser parser = new Parser("(d or b) xor not (a or c)");
parser.getExpression().accept(new ExpressionTreePrinter());
```

Ispis će biti:

```
xor
  or
    D
    B
  not
    or
      A
      C
```

iz čega je vidljivo da `xor` ima dva djeteta: čvor `or` i čvor `not`. Čvor `or` ima dva djeteta: varijable `D` i `B`. Čvor `not` ima jedno dijete: operator `or` koji ima dva djeteta: varijable `A` i `C`.

Ubacite u projekt demonstracijski program `Izrazi2.java`. Osigurajte da Vam ispitni program daje izlaz kakav je prikazan u `Izrazi2.txt`.

Problem 3.

Napišite posjetitelja `VariablesGetter` koji će odrediti sve varijable koje se spominju u izrazu i koje će vratiti kao listu naziva varijabli sortiranu leksikografski. Naravno, elementi liste moraju biti jedinstveni. Evo primjera uporabe:

```
Parser parser = new Parser("(c + a) xor (a or b)");
VariablesGetter getter = new VariablesGetter();
parser.getExpression().accept(getter);
List<String> variables = getter.getVariables();
```

Rezultat će biti lista ("A", "B", "C").

Ubacite u projekt demonstracijski program `Izrazi3.java`. Osigurajte da Vam ispitni program daje izlaz kakav je prikazan u `Izrazi3.txt`.

Problem 4.

U paket `hr.fer.zemris.bf.utils` dodajte razred `Util`. Dodajte u njega statičku metodu:

```
public static void forEach(List<String> variables, Consumer<boolean[]> consumer);
```

Zadaća ove metode jest da za zadanu listu varijabli generira redom sve kombinacije vrijednosti (kao da želite generirati tablicu istinitosti), te za svaku kombinaciju vrijednosti poziva definirani `consumer`. Vrijednosti se moraju generirati upravo redosljedom kojim biste ih slagali u tablici istinitosti: od svih nula (tj. false) prema svim jedinicama (tj. true), gdje je nabrže mijenja najdesnija varijabla (zadnji element liste).

Evo primjera:

```
Util.forEach(
            Arrays.asList("A","B","C"),
            values ->
                  System.out.println(
                        Arrays.toString(values)
                              .replaceAll("true", "1")
                              .replaceAll("false", "0")
                  )
);
```

Rezultat će biti:

```
[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
[1, 0, 0]
[1, 0, 1]
[1, 1, 0]
[1, 1, 1]
```

Ovaj primjer imate kao demonstracijski program `ForEachDemo1.java`; ubacite ga u projekt i osigurajte da Vam daje prikazani izlaz.

Problem 5.

U paket `hr.fer.zemris.bf.utils` dodajte novog posjetitelja, razred `ExpressionEvaluator`. Zadaća ovog posjetitelja jest izračun vrijednosti izraza za zadanu kombinaciju ulaznih varijabli. Pogledajmo najprije primjer uporabe.

```
Node expression = new Parser("A and b or C").getExpression();
List<String> variables = Arrays.asList("A", "B", "C");
ExpressionEvaluator eval = new ExpressionEvaluator(variables);

eval.setValues(new boolean[] {false, false, false});
expression.accept(eval);
System.out.println("f(A,B,C) = f(0,0,0) = " + eval.getResult());

eval.setValues(new boolean[] {false, false, true});
expression.accept(eval);
System.out.println("f(A,B,C) = f(0,0,1) = " + eval.getResult());
```

Što se događa u prethodnom primjeru? Stvorili smo posjetitelja i predali mu popis varijabli `A`, `B`, `C`. Potom

smo pozvali `setValues` i predali polje od tri booleove vrijednosti. Posljetitelj sada pamti da je varijabli `A` postavljena nulta vrijednost iz predanog polja, varijabli `B` prva a varijabli `C` druga (u našem slučaju, prvi poziv sve tri varijable postavlja na `false`). Konačno, posljetitelja šaljemo izrazu kroz metodu `accept`. Rezultat će biti provođenje izračuna koji potom dohvaćamo i ispisujemo.

Da bi ovo funkcioniralo, nekoliko stvari moramo složiti na korektan način.

Posjetitelju moramo predati popis varijabli, te na neki način moramo pamtiti koju vrijednost trenutno ima koja varijabla. Za ovo ćemo alocirati polje booleovih vrijednosti čija veličina odgovara broju varijabli.

Uz polje ovih vrijednosti, održavat ćemo i mapu koja ime varijable preslikava u redni broj. Primjerice, ako nam korisnik preda popis varijabli "A", "E", "Z", u mapi ćemo zapamtiti da je A bila na poziciji 0, E na poziciji 1 a Z na poziciji 2. Na tim istim pozicijama u polju booleovih vrijednosti pronaći ćemo i vrijednost koju varijabla ima dodijeljenu.

Uz ovo, za potrebe izračuna trebat ćemo i jedan stog booleovih vrijednosti.

Evo nekoliko implementacijskih detalja:

```
public class ExpressionEvaluator implements NodeVisitor {

	private boolean[] values;
	private Map<String, Integer> positions;
	private Stack<Boolean> stack = new Stack<>();

	public ExpressionEvaluator(List<String> variables) {...}

	public void setValues(boolean[] values) {...}

	public void visit(ConstantNode node) {...}
	public void visit(VariableNode node) {...}
	public void visit(UnaryOperatorNode node) {...}
	public void visit(BinaryOperatorNode node) {...}
	public void start() {...}
	public boolean getResult() {...}
}
```

Kada se posjetitelju preda novi niz vrijednosti (metoda `setValues`), iskopirat ćemo vrijednosti iz dobivenog polja u interno polje vrijednosti varijabli.

Kada posjetitelj dođe do čvora koji predstavlja konstantu, on tu konstantu zapisuje na stog. Kada dođe na čvor koji odgovara varijabli, u mapi čita koji je redni broj dodijeljen toj varijabli, i potom u internom polju vrijednosti čita vrijednost te varijable i gura je na stog. Ako mapa nema informaciju o varijabli, baciti iznimku `IllegalStateException`.

Uočite, u slučajevima kada je naš čitav izraz bio jedna konstanta ili jedna varijabla, obilazak "stabla" će rezultirati samo jednim guranjem na stog. Čitanjem tog jednog elementa sa stoga dolazimo do vrijednosti izraza i to je upravo ono što radi metoda `getResult()`. Ako na stogu nije točno jedna vrijednost, nešto ne valja i treba baciti iznimku (`IllegalStateException`). Metoda `getResult()` pri tome ne briše vrijednost sa stoga pa se može pozvati i više puta - vraćat će vrijednost zadnjeg provedenog izračuna, sve dok se stog ne obriše pozivom metode `start()` čime se stog briše i tako priprema za provođenje novog izračuna. Dopunite metodu `setValues` tako da i ona poziva metodu `start()` čime se posjetitelj automatski priprema za provođenje novog izračuna.

Ostalo je razmotriti što učiti sa čvorovima koji su operatori. Ali tu je situacija vrlo jednostavna: oni naprosto

sebe šalju svojoj djeci, čime će u konačnici za svako dijete na stogu nastati po jedna booleova vrijednost. Čvor potom sa stoga skida onoliko booleovih vrijednosti koliko ima djece, te vrijednosti kombinira predanom strategijom (sjetite se da čvorovi koji modeliraju unarne i binarne operatore u konstruktoru traže i strategiju koja zna obraditi jednu odnosno dvije vrijednosti), i konačni rezultat djelovanja operatora ponovno guraju na stog, čime je isti spreman ili za dohvat kao konačan rezultat, ili za kombiniranje u nekom od operatora koji su u stablu bili na višoj razini.

Pogledajte sada demonstracijski program `ForEachDemo2.java`; ubacite ga u projekt i osigurajte da Vam daje prikazani izlaz. Program kombinira forEach metodu koju ste napravili u zadatku 4 i posjetitelja razvijenog u zadatku 5 kako bi ispisao tablicu istinitosti proizvoljne Booleove funkcije. Pokretanjem programa morate dobiti sljedeći ispis.

```
[0, 0, 0] ==> 0
[0, 0, 1] ==> 1
[0, 1, 0] ==> 0
[0, 1, 1] ==> 1
[1, 0, 0] ==> 0
[1, 0, 1] ==> 1
[1, 1, 0] ==> 1
[1, 1, 1] ==> 1
```

Problem 6.

U paketu `hr.fer.zemris.bf.utils` u razred `Util` dodajte novu statičku metodu:

```
public static Set<boolean[]> filterAssignments(
     List<String> variables, Node expression, boolean expressionValue
);
```

Metoda prima popis varijabli i jedan izraz, stvara sve kombinacije varijabli, uporabom prethodno napisanog posjetitelja računa vrijednost funkcije, i ako se ona podudara s vrijednosti koja je predana kao treći argument ove metode, dodaje tu kombinaciju u skup kombinacija koji na kraju vraća. Skup mora biti tako podešen da mu iterator vraća elemente poretkom koji kombinacije imaju u tablici istinitosti. Primjerice, ako smo imali funkciju od dvije varijable, i ako je ona bila 1 za kombinacije [true, false] i [false, false], iterator mora najprije vratiti [false, false] a potom [true, false]. Razmislite koja Vam implementacija skupa odgovara i kako je podesiti da postignete ovakvo ponašanje.

Pogledajte sada demonstracijski program `UtilDemo1.java`; ubacite ga u projekt i osigurajte da Vam daje prikazani izlaz. Izlaz mora biti:

```
[0, 0, 1]
[0, 1, 1]
[1, 0, 1]
[1, 1, 0]
[1, 1, 1]
```

Problem 7.

U paketu `hr.fer.zemris.bf.utils` u razred `Util` dodajte još i sljedeće statičke metode.

```
public static int booleanArrayToInt(boolean[] values);
```

Metoda prima polje booleovih vrijednosti i pretvara ga u redni broj retka gdje se ta kombinacija nalazi u tablici istinitosti (numeracija ide od 0); npr. za polje `[false, false, true, true]` vraća `3`.

```
public static Set<Integer> toSumOfMinterms(
  List<String> variables, Node expression
);
public static Set<Integer> toProductOfMaxterms(
  List<String> variables, Node expression
);
```

Metode vraćaju skup brojeva koji predstavljaju minterme odnosno maksterme koje funkcija sadrži. Obje funkcije možete lijepo riješiti delegiranjem trećoj privatnoj funkciji koja se pak oslanja na `filterAssignments` i `booleanArrayToInt`.

Pogledajte sada demonstracijski program `UtilDemo2.java`; ubacite ga u projekt i osigurajte da Vam daje prikazani izlaz. Izlaz mora biti:

```
Mintermi f([A, B, C]): [1, 3, 5, 6, 7]
Mintermi f([C, B, A]): [3, 4, 5, 6, 7]
```

Primijetite da stablo kao stablo ne zna ništa o mintermima ili makstermima: stablo je samo reprezentacija izraza odnosno određuje kako se dolazi do vrijednosti uz poznato stanje varijabli. Redni brojevi koje pridružujemo mintermima i makstermima određeni su poretkom varijabli koji postavimo na funkciju – upravo to ilustrira prethodni primjer.

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries for this homework (whether it is yours old code or someones else). You can use Java Collection Framework and other parts of Java covered by lectures; if unsure – e-mail me. Document your code!

If you need any help, I have reserved a slot for consultations every day from Tuesday to Friday at 1 PM. Feel free to drop by my office.

All source files must be written using UTF-8 encoding. All classes, methods and fields (public, private or otherwise) must have appropriate javadoc.

You are expected to write tests for Lexer class methods.
You are encouraged to write tests for other problems.

When your **complete** homework is done, pack it in zip archive with name `hw07-0000000000.zip` (replace zeros with your JMBAG). Upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is April 22th 2017. at 06:00 PM.