UNIVERSITÉ
CAEN
NORMANDIE

Réponses aux questions du devoir de Maison Autres Paradigmes

21812350 Brandon VOUVOU
21812339 Faina AIT HAMMOUDA

27 Avril 2020

LICENCE 2e ANNÉE INFORMATIQUE
Groupe 1B

Partie n° 1

1 Représentation en Haskell

Réponse n° 1 **:** Définition de la fonction ***(visuFormule f)*** qui permet de visualiser une formule ***f***.

```
visuFormule :: Formule -> String
visuFormule (Var p)    = p
visuFormule (Non f)    = "~" ++ (visuFormule f)
visuFormule (Et g d)   = "(" ++ (visuFormule g) ++ " & " ++ (visuFormule d) ++ ")"
visuFormule (Ou g d)   = "(" ++ (visuFormule g) ++ " v " ++ (visuFormule d) ++ ")"
visuFormule (Imp g d)  = "(" ++ (visuFormule g) ++ " => " ++ (visuFormule d) ++ ")"
visuFormule (Equi g d) = "(" ++ (visuFormule g) ++ " <=> " ++ (visuFormule d) ++ ")"
pprint                 = putStrLn . visuFormule
```

2 Mise sous forme clausale de formule du calcul propositionnel

2.1 Mise sous forme clausale d'une formule

2.1.1 Éliminer les opérateurs Imp et Equi

Réponse n° 2 **:**

a) On peut remplacer *(Imp g d)* par *(Ou (Non g) d)* parce que par définition sous forme composé d'autres connecteurs *(Imp g d)* équivaut à ***(Ou (Non g) d)***

b) On peut remplacer *(Equi g d)* par *(Et (Imp g d) (Imp g d))* et donc par ***(Et (Ou (Non g) d) (Ou (Non g) d)))***

Réponse n° 3 **:** Définition de la fonction ***(elimine f)*** qui fait disparaître les opérateurs ***Imp*** & ***Equi*** de la formule ***f***.

```
elimine :: Formule -> Formule
elimine (Var p)    = Var p
elimine (Non f)    = Non (elimine f)
elimine (Et g d)   = Et (elimine g) (elimine d)
elimine (Ou g d)   = Ou (elimine g) (elimine d)
elimine (Imp g d)  = Ou (Non (elimine g)) (elimine d)
elimine (Equi g d) = Et (Ou (Non (elimine g)) (elimine d)) (Ou (elimine g) (Non d))
```

2.1.2 Amener les négations devant les littéraux positifs

Réponse n° 4 **:** Soit f une formule. On peut remplacer *(Non (Non f))* par ***f***.

Réponse n° 5 **:** Rappelle des deux lois de De Morgan.

1. *Première loi :* ***(Imp (Non (Et g d)) (Ou (Non g) (Non d)))*** ou $\neg$ (A$\wedge$B) $\rightsquigarrow$ $\neg$ A$\vee\neg$B
2. *Deuxième loi :* ***(Imp (Non (Ou g d)) (Et (Non g) (Non d)))*** ou $\neg$ (A$\vee$B) $\rightsquigarrow$ $\neg$ A$\wedge\neg$ B

– On définit la fonction ***(ameneNon f)*** qui amène les négations devant les littéraux positifs de la formule ***f***.

```
ameneNon:: Formule -> Formule
ameneNon (Var p) = (Var p)
ameneNon (Non f) = disNon f
ameneNon (Et g d) = (Et (ameneNon g) (ameneNon d))
ameneNon (Ou g d) = (Ou (ameneNon g) (ameneNon d))
```

Réponse n° 6 **:** La fonction (disNon f) enlève les négations sur les littéraux négatifs.

```
disNon :: Formule -> Formule
disNon (Var p)  = (Non (Var p))
disNon (Non f)  = ameneNon f
disNon (Et g d) = (Ou (disNon g) (disNon d))
disNon (Ou g d) = (Et (disNon g) (disNon d))
```

2.1.3 Faire apparaître une conjonction de clauses

```
normalise :: Formule -> Formule
normalise (Et g d) = concEt (normalise g) (normalise d)
normalise (Ou g d) = developper (normalise g) (normalise d)
normalise f        = f
```

```
concEt :: Formule -> Formule -> Formule
concEt (Et g d) f = (Et g (concEt d f))
concEt g f        = (Et g f)
```

Réponse n° 7 **:** Définition de la fonction ***(developper g d)*** (qui pourra utiliser une ou plusieurs fonctions auxiliaires)

```
        -- fonction auxiliaire
developperAux :: Formule -> Formule -> Formule
developperAux f (Et g d) = (Et (developperAux g f ) (developperAux d f))
developperAux f g = (Ou g f)


developper :: Formule -> Formule -> Formule
developper (Et g d) f = concEt (developperAux g f) (developperAux d f)
developper f (Et g d) = concEt (developperAux g f) (developperAux d f)
developper f g        = (Ou f g)
```

Réponse n° 8 **:** Déduisons-en la définition de la fonction ***(formeClausale f)*** qui applique successivement chacune des trois étapes à la formule f

```
formeClausale :: Formule -> Formule
formeClausale f = normalise (ameneNon (elimine f))
```

Partie n° 2

3 Résolvante et principe de résolution

```
type Clause = [Formule]
type FormuleBis = [Clause]
```

3.1 Transformer une Formule en une FormuleBis

Réponse n° 9 **:** Complétons les définitions des fonctions ci-dessous qui permettent de transformer une formule f (déjà sous forme clausale) en une FormuleBis, i.e. en une liste de clauses.

```
etToListe :: Formule -> FormuleBis
etToListe (Et g d) = (ouToListe g) : etToListe d
etToListe f        = [ouToListe f]

ouToListe :: Formule -> Clause
ouToListe (Ou g d) = g : (ouToListe d)
ouToListe f        = [f]
```

Réponse n° 10 **:** Définition de la fonction (neg l) qui à un littéral l associe sa négation.

```
neg :: Formule -> Formule
neg (Non l) = l
neg l       = (Non l)
```

Réponse n° 11 **:** Complétons la définition de la fonction (sontLiees xs ys) qui détermine si deux clauses xs et ys sont liées.

```
sontLiees :: Clause -> Clause -> Bool
sontLiees [] _ = False
sontLiees _ [] = False
sontLiees (x:xs) (y:ys)
  | x == (neg y) = True
  | sontLiees xs (y:ys) == False = sontLiees (x:xs) ys
  | sontLiees (x:xs) ys == False = sontLiees xs (y:ys)
  | otherwise = True
```

Réponse n° 12 **:** Complétons la définition de la fonction (resolvante xs ys) qui calcule une résolvante des deux clauses xs et ys qui sont supposées être liées

```
        -- fonction auxiliaire
estDansClause :: Formule -> Clause -> Bool
estDansClause _ [] = False
estDansClause x (y:ys)
  | x == y = True
  | otherwise = estDansClause x ys
```

```haskell
resolvante :: Clause -> Clause -> Clause
resolvante [] _ = []
resolvante _ [] = []
resolvante (x:xs) (y:ys)
  | x == (neg y) = xs ++ ys
  | x == y = x:(resolvante xs ys)
  | otherwise = if (estDansClause x(resolvante xs (y:ys)))
    then (resolvante xs (y:ys))
    else (x:(resolvante xs (y:ys)))
```

************************* Partie Bonus *******************************

```haskell
deduire :: Formule -> Clause
deduire x = resoudre (head sorite) (tail sorite)
    where sorite = (formeClausaleBis (formeClausale x))

resoudre :: Clause -> FormuleBis -> Clause
resoudre xs [] = xs
resoudre xs (ys:yss)
  | sontLiees xs ys  = resoudre (resolvante xs ys) yss
  | otherwise  = resoudre xs (yss ++ [ys])
```