Software Testing Assignment 3

Cindy Berghuizen, Omar Pakker , Chiel Peter, Maria Gouseti

22 September , 2013

Exercise 5

Properties

The testable properties we defined and used in our version of *isPermutation* are (see figure 1):

- an empty list is a permutation of an empty list
- both lists have the same length (the first list is empty and the second one still has elements)
- A list L' is a permutation of a list L if every appearance of an element that exists in list L also exists in L' and L' does not have other elements.

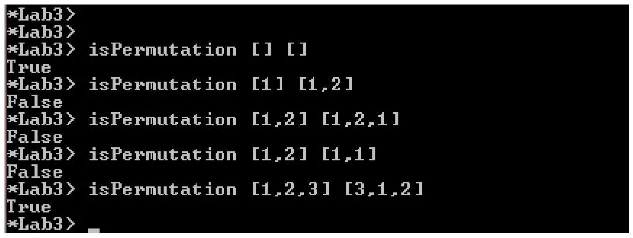
Testing

In order to test *isPermutation* we are going to use *permutations* a function from Haskell Data.List. This function gets a list as an argument and returns a list we all its permutations, including the initial list as well. Consequently we check if the the second argument of *isPermutation* is an element of the list created by *permutations* with argument the first list. If both functions have the same value *isPermutation* is correct. We also created *testPermutationsTotal* which presents if all the test were successful (see figure 2).

```
--Exercise 5
testPermutation :: IO Bool
testPermutation = do
                              p1 <- genIntList
                              p2 <- genIntList
                              return ((isPermutation p1 p2) == (elem p2 (permutations p1)))

testPermutations :: Int -> IO [Bool]
testPermutations 0 = return []
testPermutations c = do
                              p <- testPermutation
                              ps <- testPermutations (c-1)
                              return (p:ps)

testPermutationsTotal :: Int -> IO String
testPermutationsTotal c = do
                              ps <- testPermutations c
                              return ("All Checks Valid: " ++ (show (all (\x -> x) ps)))
```

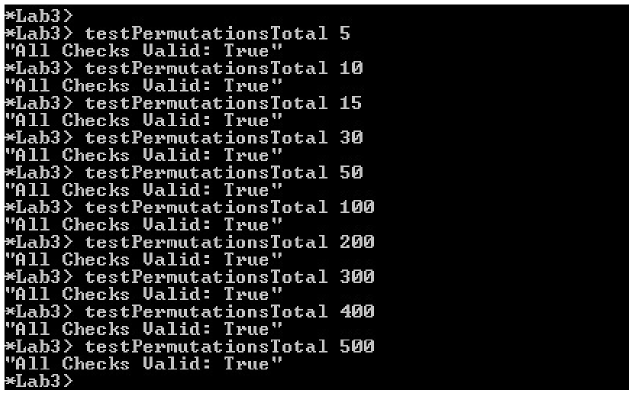
Exercise 6

When we started testing our CNF converter using our random formula generator we noticed that we have missed the case of an empty disjunction as a result in this case the function did not work properly. So we added a line to catch this case in order to test the function. The fixed code of our CNF converter:



```
*Lab3>
*Lab3>
*Lab3> isPermutation [] []
True
*Lab3> isPermutation [1] [1,2]
False
*Lab3> isPermutation [1,2] [1,2,1]
False
*Lab3> isPermutation [1,2] [1,1]
False
*Lab3> isPermutation [1,2,3] [3,1,2]
True
*Lab3> _
```

Figure 1: Manual execution of *isPermutation* testing the properties mentioned above.

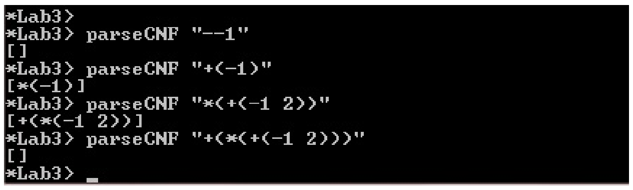


```
*Lab3>
*Lab3> testPermutationsTotal 5
"All Checks Valid: True"
*Lab3> testPermutationsTotal 10
"All Checks Valid: True"
*Lab3> testPermutationsTotal 15
"All Checks Valid: True"
*Lab3> testPermutationsTotal 30
"All Checks Valid: True"
*Lab3> testPermutationsTotal 50
"All Checks Valid: True"
*Lab3> testPermutationsTotal 100
"All Checks Valid: True"
*Lab3> testPermutationsTotal 200
"All Checks Valid: True"
*Lab3> testPermutationsTotal 300
"All Checks Valid: True"
*Lab3> testPermutationsTotal 400
"All Checks Valid: True"
*Lab3> testPermutationsTotal 500
"All Checks Valid: True"
*Lab3>
```

Figure 2: *testPermutationsTotal*

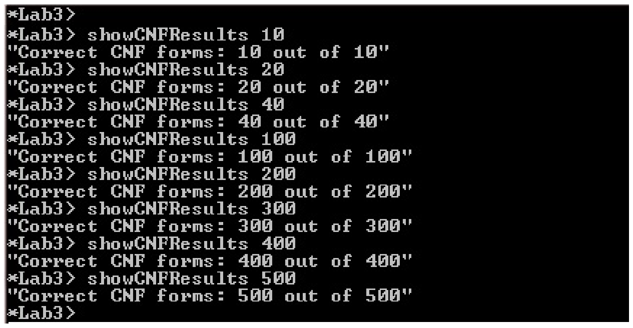
```
-- Precondition: Form is arrowfree and in negative normal form
-- Postcondition: Form is in conjunctive normal form
cnf :: Form -> Form
cnf (Prop x)            = Prop x
cnf (Neg (Prop x))      = Neg (Prop x)
cnf (Cnj f)             = Cnj (map cnf f)
cnf (Dsj []) = Dsj [] --Added 2013-09-16; previously missed
cnf (Dsj [f, g])        = dist (cnf f) (cnf g)
cnf (Dsj (f:fs))        = dist (cnf f) (cnf (Dsj fs))

-- Precondition: Forms are in conjunctive normal form
-- Postcondition: Form is the the conjunctive normal form of (form1 v form2)
dist :: Form -> Form -> Form
dist (Cnj fs) g         = Cnj (map (dist g) fs)
dist f (Cnj gs)         = Cnj (map (dist f) gs)
dist f g                = Dsj [f,g]
```

To test the CNF converter we used two functions that approach it differently. The first one is *equiv* from assignment 2. With this function we can test that the original form and its CNF version are equivalent (*testCNF*). Then we changed the parser of predicate logic to accept only forms in CNF (*parseCNF* as shown in figure 3). If the result of our CNF converter passes both tests then the CNF converter is correct. The function *showCNFResults* returns the score of the correct conversions (see figure 4).



```
*Lab3>
*Lab3> parseCNF "--1"
[]
*Lab3> parseCNF "+(-1)"
[*(-1)]
*Lab3> parseCNF "*(+(-1 2))"
[+(*(-1 2))]
*Lab3> parseCNF "+(*(+(-1 2)))"
[]
*Lab3> _
```

Figure 3: *parseCNF* returns [] when its argument is not in CNF.



```
*Lab3>
*Lab3> showCNFResults 10
"Correct CNF forms: 10 out of 10"
*Lab3> showCNFResults 20
"Correct CNF forms: 20 out of 20"
*Lab3> showCNFResults 40
"Correct CNF forms: 40 out of 40"
*Lab3> showCNFResults 100
"Correct CNF forms: 100 out of 100"
*Lab3> showCNFResults 200
"Correct CNF forms: 200 out of 200"
*Lab3> showCNFResults 300
"Correct CNF forms: 300 out of 300"
*Lab3> showCNFResults 400
"Correct CNF forms: 400 out of 400"
*Lab3> showCNFResults 500
"Correct CNF forms: 500 out of 500"
*Lab3>
```

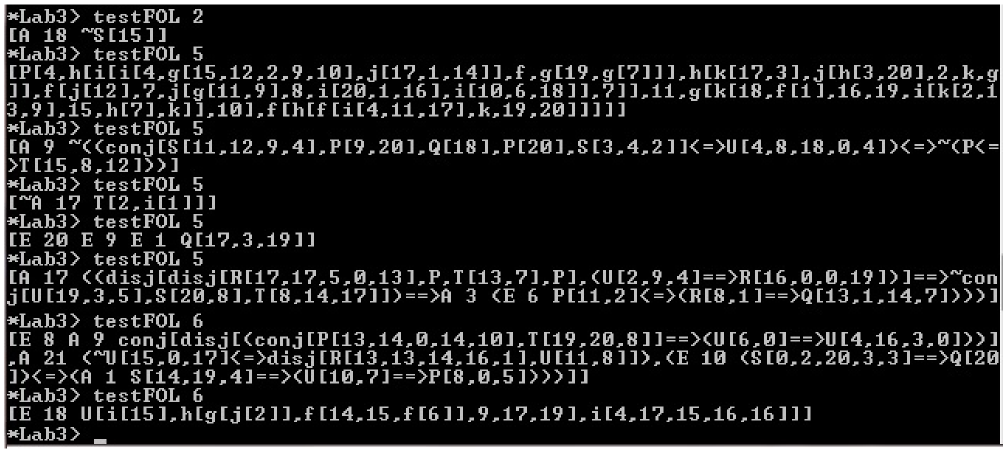
Figure 4: *showCNFResults*

```
showCNFResults n = do
                      r <- (testCNFs n)
                      return ("Correct CNF forms: "++(show (length (filter ((==)
                                True) r)))++" out of "++(show (length r)))

testCNFs n = do
                      g <- (getRandomFs n)
```

```haskell
                      return (map ( \x -> testCNF x) g)

testCNF f = (equiv f g) && ((parseCNF (formToString g))/=[]) where g = (cnf (nnf f))

formToString :: Form -> String
formToString form = show form

parseCNFForm :: Int->(Parser Token Form)
parseCNFForm i (TokenInt x: tokens) = [(Prop x,tokens)]
parseCNFForm i (TokenNeg: TokenInt x : tokens) = [ (Week2.Neg (Prop x), tokens)]
parseCNFForm i (TokenCnj : TokenOP : tokens) | i==0 = [ (Dsj fs, rest) | (fs,rest) <- parseCNFForms i tokens ]
                                             | otherwise = []
parseCNFForm i (TokenDsj : TokenOP : tokens) = [ (Cnj fs, rest) | (fs,rest) <- parseCNFForms (i+1) tokens ]
parseCNFForm i tokens = []

parseCNFForms :: Int-> (Parser Token [Form])
parseCNFForms i (TokenCP : tokens) = succeed [] tokens
parseCNFForms i tokens = [(f:fs, rest) | (f,ys) <- parseCNFForm i tokens, (fs,rest) <- parseCNFForms i ys ]

parseCNF :: String -> [Form]
parseCNF s = [ f | (f,_) <- parseCNFForm 0 (lexer s) ]
```

Exercise 8 (Bonus Exercise)

We created a parser for first order logic as it was described in Week3.hs with a small change in the case of infix operator "==". We changed *show* to put the operator and its terms in parentheses exactly as "==>" and "<=>" are expressed. The function *testFOL* uses the random Fomula generator to acquire a Formula and then tests it with the parser. The argument of *testFOL* is the depth of *getRandomFormula*.



```
*Lab3> testFOL 2
[A 18 ~S[15]]
*Lab3> testFOL 5
[P[4,h[i[i[4,g[15,12,2,9,10],j[17,1,14]],f,g[19,g[7]]],h[k[17,3],j[h[3,20],2,k,g
]],f[j[12],7,j[g[11,9],8,i[20,1,16],i[10,6,18]],7]],11,g[k[18,f[1],16,19,i[k[2,1
3,9],15,h[7],k]],10],f[h[f[i[4,11,17],k,19,20]]]]]]
*Lab3> testFOL 5
[A 9 ~((conj[S[11,12,9,4],P[9,20],Q[18],P[20],S[3,4,2]]<=>U[4,8,18,0,4])<=>~(P<=
>T[15,8,12]))]
*Lab3> testFOL 5
[~A 17 T[2,i[1]]]
*Lab3> testFOL 5
[E 20 E 9 E 1 Q[17,3,19]]
*Lab3> testFOL 5
[A 17 ((disj[disj[R[17,17,5,0,13],P,T[13,7],P],(U[2,9,4]==>R[16,0,0,19])]==>~con
j[U[19,3,5],S[20,8],T[8,14,17]])==>A 3 (E 6 P[11,2]<=>(R[8,1]==>Q[13,1,14,7])))]
*Lab3> testFOL 6
[E 8 A 9 conj[disj[(conj[P[13,14,0,14,10],T[19,20,8]]==>(U[6,0]==>U[4,16,3,0]))]
,A 21 (~U[15,0,17]<=>disj[R[13,13,14,16,1],U[11,8]]),(E 10 (S[0,2,20,3,3]==>Q[20
])<=>(A 1 S[14,19,4]==>(U[10,7]==>P[8,0,5])))]]
*Lab3> testFOL 6
[E 18 U[i[15],h[g[j[2]],f[14,15,f[6]],9,17,19],i[4,17,15,16,16]]]
*Lab3> _
```

Figure 5: *testFOL*