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:

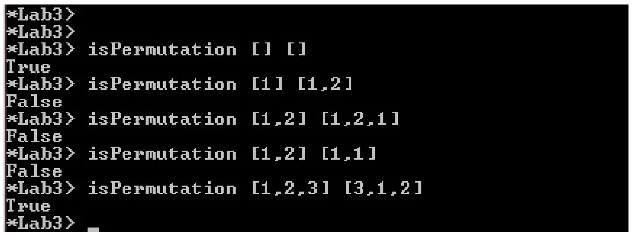

Figure 1: Manual execution of *isPermutation* testing the properties mentioned above.

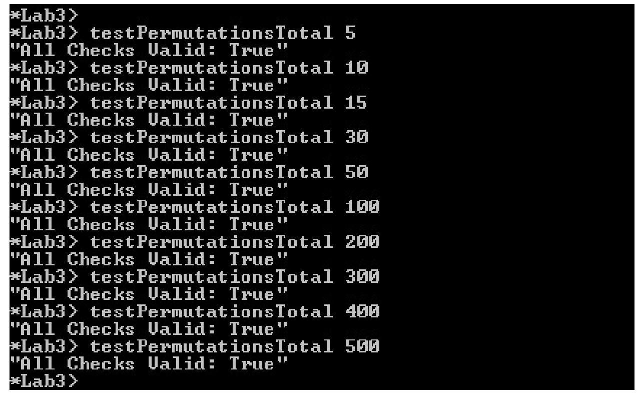

Figure 2: *testPermutationsTotal*

```haskell
-- 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).

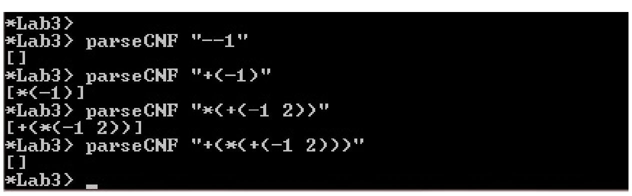

Figure 3: *parseCNF* returns [] when its argument is not in CNF.

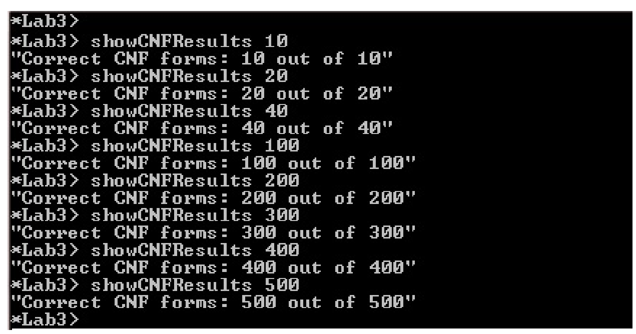

Figure 4: *showCNFResults*

```haskell
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)
```

```
                    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 parenthesis 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*.

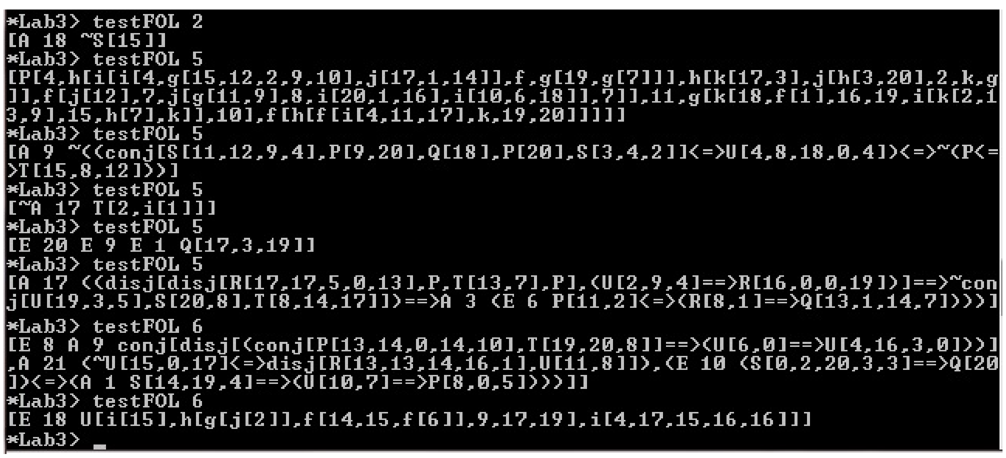

Figure 5: *testFOL*