Lab Assignments

A general note

Grading

- Grading starts from zero
  - Nothing handed in, nothing gained
- MVP: 7.5
  - Working implementation
  - Basic testing
- Extra
  - Exhaustiveness of testing strategies
  - Efficient implementation

Tips

- Heed the assignment instructions
  - And verify your solution actually covers them
  - Use your own test to find mismatches with specification
- Testing (perfect recipe)
  - Analyze possible strategies
  - Pick based on coverage of output and input space
  - Implement chosen strategies

Lecture 3 – Haskell Code

A general overview

Logical Operators Precedence [1]

1. Not
2. And
3. Or
4. Implies
5. Equivalence

$$\neg(p \wedge q) \vee r \to s$$

$$((\neg(p \wedge q)) \vee r) \to s$$

[1] O'Donnell, John; Hall, Cordelia; Page, Rex (2007), Discrete Mathematics Using a Computer, Springer, p. 120, ISBN 9781846285981.

Propositional logic to Haskell definition

$$\neg(p \wedge q) \vee r \rightarrow s$$

```haskell
data Form = Prop Name
          | Neg  Form
          | Cnj [Form]
          | Dsj [Form]
          | Impl Form Form
          | Equiv Form Form
          deriving (Eq,Ord)
type Name = Int
```

```haskell
Impl (Cnj [Neg (Dsj [Prop 1, Prop 2]), Prop 3]) (Prop 4)
```

Propositional logic to Haskell definition

$$\neg(1 \wedge 2) \vee 3 \rightarrow 4$$

```
data Form = Prop Name
          | Neg  Form
          | Cnj [Form]
          | Dsj [Form]
          | Impl Form Form
          | Equiv Form Form
         deriving (Eq,Ord)
type Name = Int
```

```
Impl (Cnj [Neg (Dsj [Prop 1, Prop 2]), Prop 3]) (Prop 4)
```

A Show instance for Form

```
>show Impl (Cnj [Neg (Dsj [Prop 1, Prop 2]), Prop 3]) (Prop 4)
(*(-+(1 2) 3)==>4)
```

A Show instance for Form

```
>show Impl (Cnj [Neg (Dsj [Prop 1, Prop 2]), Prop 3]) (Prop 4)
(*(-+(1 2) 3)==>4)
```

```
instance Show Form where
  show (Prop x)      = show x
  show (Neg f)       = '-' : show f
  show (Cnj fs)        = "*(" ++ showLst fs ++ ")"
  show (Dsj fs)        = "+(" ++ showLst fs ++ ")"
  show (Impl f1 f2)    = "(" ++ show f1 ++ "==>"
                             ++ show f2 ++ ")"
  show (Equiv f1 f2)    = "(" ++ show f1 ++ "<=>"
                             ++ show f2 ++ ")"
```

A Parser for Form

```
"(*(-+(1 2) 3)==>4)"
```

```haskell
parse :: String -> [Form]
parse s = [ f | (f,_) <- parseForm (lexer s) ]
```

A Parser for Form

```
"(*(-+(1 2) 3)==>4)"
```

```
Impl (Cnj [Neg (Dsj [Prop 1, Prop 2]), Prop 3]) (Prop 4)
```

```haskell
parse :: String -> [Form]
parse s = [ f | (f,_) <- parseForm (lexer s) ]
```

A Parser for Form

```
"(*(-+(1 2) 3)==>4)"

Impl (Cnj [Neg (Dsj [Prop 1, Prop 2]), Prop 3]) (Prop 4)

parse :: String -> [Form]
parse s = [ f | (f,_) <- parseForm (lexer s) ]

lexer :: String -> [Token]
```

A Parser for Form

```
"(*(-+(1 2) 3)==>4)"

Impl (Cnj [Neg (Dsj [Prop 1, Prop 2]), Prop 3]) (Prop 4)

parse :: String -> [Form]
parse s = [ f | (f,_) <- parseForm (lexer s) ]

lexer :: String -> [Token]

parseForm :: [Token] -> [(Form,[Token])]
```

```haskell
parseForm :: [Token] -> [(Form,[Token])]
```

```haskell
type Parser a b = [a] -> [(b,[a])]
```

```haskell
parseForm :: Parser Token Form
```

Lexer

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs) | isSpace c = lexer cs
             | isDigit c = lexNum (c:cs)
lexer ('(':cs) = TokenOP : lexer cs
lexer (')':cs) = TokenCP : lexer cs
lexer ('*':cs) = TokenCnj : lexer cs
lexer ('+':cs) = TokenDsj : lexer cs
lexer ('-':cs) = TokenNeg : lexer cs
lexer ('=':'=':'>':cs) = TokenImpl : lexer cs
lexer ('<':'=':'>':cs) = TokenEquiv : lexer cs
lexer (x:_) = error ("unknown token: " ++ [x])
```

```haskell
parseForm :: Parser Token Form
parseForm (TokenInt x: tokens) = [(Prop x,tokens)]
parseForm (TokenNeg : tokens) =
  [ (Neg f, rest) | (f,rest) <- parseForm tokens ]
parseForm (TokenCnj : TokenOP : tokens) =
  [ (Cnj fs, rest) | (fs,rest) <- parseForms tokens ]
parseForm (TokenDsj : TokenOP : tokens) =
  [ (Dsj fs, rest) | (fs,rest) <- parseForms tokens ]
parseForm (TokenOP : tokens) =
  [ (Impl f1 f2, rest) | (f1,ys) <- parseForm tokens,
                          (f2,rest) <- parseImpl ys ]
   ++
  [ (Equiv f1 f2, rest) | (f1,ys) <- parseForm tokens,
                           (f2,rest) <- parseEquiv ys ]
parseForm tokens = []
```

Arrowfree

```
arrowfree :: Form -> Form
arrowfree (Prop x) = Prop x
arrowfree (Neg f) = Neg (arrowfree f)
arrowfree (Cnj fs) = Cnj (map arrowfree fs)
arrowfree (Dsj fs) = Dsj (map arrowfree fs)
arrowfree (Impl f1 f2) =
  Dsj [Neg (arrowfree f1), arrowfree f2]
arrowfree (Equiv f1 f2) =
  Dsj [Cnj [f1', f2'], Cnj [Neg f1', Neg f2']]
  where f1' = arrowfree f1
        f2' = arrowfree f2
```

$$p \to q \leftrightarrow \neg p \vee q$$

$$(p \leftrightarrow q) \leftrightarrow (p \wedge q) \vee (\neg p \wedge \neg q)$$

Negation Normal Form

```haskell
nnf :: Form -> Form
nnf (Prop x) = Prop x
nnf (Neg (Prop x)) = Neg (Prop x)
nnf (Neg (Neg f)) = nnf f
nnf (Cnj fs) = Cnj (map nnf fs)
nnf (Dsj fs) = Dsj (map nnf fs)
nnf (Neg (Cnj fs)) = Dsj (map (nnf.Neg) fs)
nnf (Neg (Dsj fs)) = Cnj (map (nnf.Neg) fs)
```

$$\neg(\neg p) \leftrightarrow p$$

$$\neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$$

$$\neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$$

That’s it

(not really)