Lab 11 - Expression Trees and Term Rewriting

CS 1XA3

March 26^{th}, 2018

Recursive Data Types In Haskell

- By now, you should have the hang of how to define your own List type in Haskell

```haskell
data List a = Node a (List a) | Empty
```

- Recursive data types allow you to define linked data structures
- By linking values together in different ways we can model different relationships between data

Binary Trees (1)

- A Binary Tree is like a Linked List with two links at each Node instead of one

- There are a few different approaches to creating Binary Trees you could take, consider the following example that holds all its values in each Leaf

```haskell
data BinTree a = Node (BinTree a) (BinTree a)
               | Leaf a

-- Example
Node (Node (Leaf 1) (Leaf 2))
     (Node (Leaf 3) (Leaf 4))
```

Binary Trees (2)

- Alternatively, we could keep values inside the Nodes as well as each Leaf

```
data BinTree a = Node a (BinTree a) (BinTree a)
               | Leaf a

-- Example
Node 1 (Node 2 (Leaf 3) (Leaf 4))
       (Leaf 5)
```

- Even more choices: Perhaps we use Empty instead of Leaf a, or we give the option of Empty or Leaf. Challenge: try implementing those Trees

Multi-Way Trees (Rose Trees)

- Multi-way Trees (sometimes called Rose Trees), allow for an arbitrary number of branches at each Node (as opposed to the strict 2 in Binary Trees)
- How would we implement such a tree?

Multi-Way Trees (Rose Trees)

- Multi-way Trees (sometimes called Rose Trees), allow for an arbitrary number of branches at each Node (as opposed to the strict 2 in Binary Trees)
- How would we implement such a tree?

```
data RoseTree a = RoseTree a [RoseTree a]
```

- Food for thought: why are there no leaves? What if we want an empty tree?

Data Maps (Dictionaries)

- Maps (also known as Dictionaries) associate key/value pairs, allowing indexing of values by keys

- Efficient definitions of Maps are a bit tricky, luckily a very efficient one is provided by Data.Map.Strict.

```
data Map k v = ...
```

- It's recommended to import with a qualifier, i.e

```
import qualified Data.Map.Strict as Map
```

Using Data Maps

- You can construct a Map from a list of tuples like using the fromList function

```
import qualified Data.Map.Strict as Map

intMap :: Map.Map Int String
intMap = Map.fromList [(0,"Hello"),(1,"GoodBye")]
```

- Retrieve a value from a key using the lookup function

```
zeroLookup :: Map.Map Int b -> b
zeroLookup intMap = case Map.lookup 0 intMap of
          Just val -> val
          Nothing  -> error "Error: lookup failed"
```

Expression Trees

- The previous tree's had generalized nodes (i.e indistinguishable from eachother).
- Sometimes we wish to encode more specific information about each Node, for example when encoding an expression
- The following tree structure can be used to encode expressions with operators of different arity (# of arguments)

```
data Expr a = Op1 (Expr a) (Expr a)
            | Op2 (Expr a) (Expr a) (Expr a)
            | Op3 (Expr a)
            | Const a
```

Example: Boolean Expressions

- Consider the following expression tree for encoding boolean expressions

```haskell
data BExpr a = And (BExpr a) (BExpr a)
             | Or (BExpr a) (BExpr a)
             | Not (BExpr a)
             | Const a
             | Var String
```

- Common boolean expressions can be encoded with this type like so

```haskell
-- (true or false) and (not false)
expr :: BExpr Bool
expr = And (Or (Const True) (Const False))
           (Not (Const False))
```

Evaluating Encoded Expressions

- We can evaluate an expression of type Bexpr Bool to Bool by pattern matching

```
eval (And e1 e2) = (eval e1) && eval e2
eval (Const x) = x
...
```

- For evaluating Var's, we need to provide values for given identifiers. We can use a Map

```
eval :: Map.Map String Bool -> BExpr Bool -> Bool
eval vrs (Var nm) = case Map.lookup nm vrs of
        (Just val) -> val
        Nothing -> error "Error: failed lookup"
```

Term Rewriting

Consider the following "simplifications" that can be performed on boolean expressions. If performed completely, the resulting expression is gaurenteed to satisfy certain conditions known as being in Conjunctive Normal Form https://en.wikipedia.org/wiki/Conjunctive_normal_form

```
-- Double Negation
Not (Not e) => e

-- De Morgans Laws
Not (Or e1 e2) => And (Not e1) (Not e2)
Not (And e1 e2) => Or (Not e1) (Not e2)

-- Distributivity
Or e1 (And e2 e3) => And (Or e1 e2) (Or e1 e3)
Or (And e1 e2) e3 => And (Or e1 e3) (Or e2 e3)
```

Term Rewriting

We can construct a function for implementing these rules in Haskell as follows: Note: the function is unfinished

```
cnf :: BExpr Bool -> BExpr Bool
-- Double Negation
cnf (Not (Not e))  = cnf e
-- De Morgans Laws
cnf (Not (Or e1 e2))
          = cnf $ And (Not e1) (Not e2)
cnf (Not (And e1 e2))
          = cnf $ Or (Not e1) (Not e2)
-- Distributivity
cnf (Or e1 (And e2 e3))
          = cnf $ And (Or e1 e2) (Or e1 e3)
cnf (Or (And e1 e2) e3)
          = cnf $ And (Or e1 e3) (Or e2 e3)
```

Term Rewriting

In order to completely rewrite our expression to CNF, we need to cover the rest of our cases and recurse through appropriately

```
cnf (And e1 e2) = And (cnf e1) (cnf e2)
cnf (Or e1 e2)  = Or  (cnf e1) (cnf e2)
cnf (Not e)     = Not (cnf e)
cnf e           = e
```

Challenge: Numerical Expressions

- Create an datatype for numerical expressions, it should include variables with string identifiers
- Create an evaluation function for your datatype
- Write a simplification function that performs the following simplifications
  - $0 + x = x + 0 = 0$
  - $x - x = 0$
  - $0 * x = x * 0 = 0$
  - $x^i * x^j = x^{i+j}$

Hint: implement exponents with the tag Exp (Expr a) a