CS 456/556: Advanced Declarative Programming
Flipped Classroom Exercise 6

1. Recall from lecture that functions *car* and *cdr* can be defined as follows:

```
car :: [a] -> Maybe a
car (x:xs) = Just x
car [] = Nothing

cdr :: [a] -> Maybe [a]
cdr (x:xs) = Just xs
car [] = Nothing

cadddr ::  [a] -> Maybe a
cadddr = cdr >=> cdr >=> cdr >=> car
```

   Use the *MaybeT* monad transformer to define functions *kar* and *kdr* which work just like *car* and *cdr* except that they also print their return value if it is not *Nothing*. For example,

```
> runMaybeT $ kadddr ['a'..'z']
"bcdefghijklmnopqrstuvwxyz"
"cdefghijklmnopqrstuvwxyz"
"defghijklmnopqrstuvwxyz"
'd'
Just 'd'
```

   Hint: *kar* and *kdr* should have the following type signatures:

```
kar :: Show a => [a] -> MaybeT IO a
kdr :: Show a => [a] -> MaybeT IO [a]
```

   You'll also need to import *MaybeT* as follows

```
import Control.Monad.Trans.Maybe
```

2. Bill is a rich old man who hates cats. He will donate 1 million dollars to UNM if and only if he can be sure that none of his grandchildren own cats. Bill makes a list of people in his family and their children, beginning with himself:

```
data Person = Bill | Bob | Amy | Val | Jim | Kim deriving (Show, Eq)

ks = [(Bill, [Bob, Amy]), (Bob, [Val, Jim]), (Amy, []), (Val, [Kim])]
```

Bill doesn't know if Kim has kids because he is a bad man. Consequently, Kim is not on the list. Observe that not knowing if someone has a kid, *e.g.,* Kim, is different than knowing someone doesn't, *e.g.,* Amy.

```
kids :: Person -> ListT Maybe Person
kids = ListT . flip lookup ks
```

Bill tries to remember what kinds of pets people in his family own. He makes another list:

```
data Pet = Cat | Dog deriving (Show, Eq)

ps = [(Bill, [Dog]), (Bob, [Cat, Dog]), (Val, []), (Kim, [Cat])]
```

Bill doesn't know if Jim has any pets because Bill is a bad man. Consequently, Jim is not on the list. Observe that not knowing if someone has a pet, *e.g.,* Jim, is different than knowing someone doesn't, *e.g.,* Val.

```
pets :: Person -> ListT Maybe Pet
pets = ListT . flip lookup ps
```

Write a function *grandkidscats* with type signature *Person −> Maybe Bool* which when applied to *Bill* returns *Just True* if UNM will get 1 million dollars, *Just False* if it won't, and *Nothing* if there is not enough information. Hint: You'll need to import *ListT* as follows

```
import Control.Monad.Trans.List
```

3. Consider the following definition of a parser combinator type synonym:

```
type Parser a = StateT String Maybe a
```

   - Define a *Parser Char* called *item* that consumes a character.
   - Define a function called *sat* that takes a predicate *Char −> Bool* argument and returns a *Parser Char* that consumes a character if it satisfies the predicate and fails otherwise.
   - Define a function called *string* that takes a *String* argument and returns a *Parser String* that consumes a matching string and fails otherwise.

4. Consider the following definition of a parser combinator type synonym:

```
type Dictionary = [String]
type Parser' a = ReaderT Dictionary (StateT String Maybe) a
```

   - Define a *Parser' Char* called *item* that consumes a character.

- Define a function called *sat* that takes a predicate *Char −> Bool* argument and returns a *Parser' Char* that consumes a character if it satisfies the predicate and fails otherwise.
- Define a function called *string* that takes a *String* argument and returns a *Parser' String* that consumes a matching string and fails otherwise.
- Define a *Parser' String* called *word* which consumes strings in its dictionary and fails for other strings.

5. Consider the following definition of a binary tree type:

```
data Btree a = Leaf a | Fork (Btree a) (Btree a) deriving Show
```

The derived instance for *Show* will cause a *Btree* to be printed as Haskell code which could be used to construct it. For example,

```
> Fork (Leaf 1) (Fork (Fork (Leaf 2) (Leaf 3)) (Leaf 4))
Fork (Leaf 1) (Fork (Fork (Leaf 2) (Leaf 3)) (Leaf 4))
```

Define a function *btree :: (Read a, Show a) => [a] −> String −> Btree a* that will construct a *Btree a* from its printed representation. The parser should fail if it encounters a word that is not in its dictionary. For example,

```
data Beatle = John | Paul | George | Ringo
  deriving (Show, Eq, Enum, Bounded, Read)

beatles = [(John)..(Ringo)]

> btree beatles "Fork (Leaf John) (Fork (Fork (Leaf Paul) (Leaf George))
  (Leaf Ringo))"
Fork (Leaf John)(Fork (Fork (Leaf Paul)(Leaf George))(Leaf Ringo))
```