Continuation Passing Style

We beginnen met een waarschijnlijk nog ongekende feature te introduceren. Met onderstaande syntax kunnen we speciale features van de compiler aanzetten. Deze zijn echter niet altijd even stabiel of verspreid onder de verschillende compilers, dus we proberen deze te vermijden. Het nut van deze feature wordt onderaan duidelijker.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
```

We importeren enkele modules.

```
import Control.Monad.Cont
import Control.Monad.State
import Control.Applicative
```

Exceptions in Continuation Passing Style

Met de *flow control*-krachten van CPS kunnen we een vorm van uitzonderingen implementeren in Haskell binnen monads, gelijkaardig aan de uitzonderingen die we kennen van imperatieve talen.

Gegeven volgende functie `tryCont` en de definitie van `SqrtException`.

```
tryCont :: MonadCont m => ((err -> m a) -> m a) -> (err -> m a) -> m a
tryCont c handle = callCC $ \ok -> do
                        err <- callCC $ \notOk -> do
                            x <- c notOk
                            ok x
                        handle err

data SqrtException = LessThanZero deriving (Show, Eq)
```

Schrijf een functie `sqrtIO` en een main functie die er gebruik van maakt. De `sqrtIO` functie vraagt via IO aan de gebruiker om een getal in te geven. Als dat getal positief is, print de functie de wortel van het getal uit. Is het getal negatief, gooit hij een `LessThanZero` uitzondering op:

```
ghci> :main
Geef een getal: 4
2.0
ghci> :main
Geef een getal: -1
LessThanZero
```

```
sqrtIO = undefined
```

```
main = undefined
```

Coroutines in Continuation Passing Style

De *control flow*-krachten van CPS zijn echter nog niet uitgeput hiermee. We kunnen er zelfs Coroutines mee implementeren. We definiëren een `CoroutineT` monad transformer, die zelf een combinatie is van de `ContT` en `StateT` monad transformers. De `ContT` wordt gebruikt voor de *control flow*, terwijl de `StateT` gebruikt wordt om de niet-uitvoerende coroutines bij te houden.

```
newtype CoroutineT r m a = CoroutineT
    { runCoroutineT' :: ContT r (StateT [CoroutineT r m ()] m) a
    } deriving (Functor, Applicative, Monad, MonadCont, MonadIO)
```

We maken hier gebruik van de `GeneralizedNewtypeDeriving` feature om automatisch heel wat instanties af te leiden en zo boilerplate code te vermijden. Als extra oefening op monad transformers kan je deze instanties altijd eens zelf implementeren.

Met behulp van `getCCs` en `putCCs` kunnen we de gepauzeerde coroutines manipuleren.

```
getCCs :: Monad m => CoroutineT r m [CoroutineT r m ()]
getCCs = CoroutineT $ lift get

putCCs :: Monad m => [CoroutineT r m ()] -> CoroutineT r m ()
putCCs = CoroutineT . lift . put
```

Zo zijn we klaar voor het definiëren van `dequeue` en `queue`, welke respectievelijk een gepauzeerde coroutine van de kop van de *queue* weggooien, of een gegeven coroutine in de *queue* zullen plaatsen.

```
dequeue :: Monad m => CoroutineT r m ()
dequeue = undefined

queue :: Monad m => CoroutineT r m () -> CoroutineT r m ()
queue = undefined
```

Vervolgens implementeren we `yield` en `fork`. `yield` zal de huidige coroutine in de *queue* plaatsen en verder gaan met de eerste coroutine in de *queue*. `fork` zal een gegeven nieuwe coroutine opstarten.

```
yield :: Monad m => CoroutineT r m ()
yield = undefined

fork :: Monad m => CoroutineT r m () -> CoroutineT r m ()
fork nc = undefined
```

Tenslotte krijgen jullie nog `exhaust` en een manier om je coroutines uit te voeren cadeau. Met behulp van `exhaust` kunnen we alle overblijvende coroutines afwerken. `runCoroutineT` zal een coroutine uitvoeren in de basismonad die we transformeren.

```haskell
exhaust :: Monad m => CoroutineT r m ()
exhaust = do exhausted <- null <$> getCCs
             if not exhausted
                then yield >> exhaust
                else return ()

runCoroutineT :: Monad m => CoroutineT r m r -> m r
runCoroutineT = flip evalStateT []
              . flip runContT return
              . runCoroutineT'
              . (<* exhaust)

printOne :: Int -> CoroutineT r IO ()
printOne n = do liftIO (print n)
                yield

example :: IO ()
example = runCoroutineT $ do fork $ replicateM_ 3 (printOne 3)
                             fork $ replicateM_ 4 (printOne 4)
                             replicateM_ 2 (printOne 2)
```