Madala taseme sisend/väljund

- Prelüüdi failide töölemisfunktsioonid opereerivad terve failiga ühekorraga

```
readFile      :: FilePath -> IO String
writeFile     :: FilePath -> String -> IO ()
appendFile    :: FilePath -> String -> IO ()
```

- Kui on tarvis opereerida sama failiga palju kordi üksteise järel, siis on see ebaefektiivne, kuna iga kord avatakse ja suletakse fail uuesti
- Standardteek `IO` defineerib madalama taseme sisend/väljund operatsioonid, mis võimaldavad “täpsemat” kontrolli failidega opereerimisel

Madala taseme sisend/väljund

- Failide avamine ja sulgemine:

```
openFile    :: FilePath -> IOMode -> IO Handle
hClose      :: Handle -> IO ()
data IOMode  = ReadMode   | WriteMode
             | AppendMode | ReadWriteMode
```

- Operatsioone “loetava” failiga

```
hGetChar     :: Handle -> IO Char
hGetLine     :: Handle -> IO String
hGetContents :: Handle -> IO String
```

- Operatsioone “kirjutatava” failiga

```
hPutChar  :: Handle -> Char -> IO ()
hPutStr   :: Handle -> String -> IO ()
```

Madala taseme sisend/väljund

- Eeldefineeritud sisend/väljund kanalid:

```
stdin, stdout, stderr :: Handle
```

- Veatöötlus:

```
ioError   ::  IOError -> IO a
catch     ::  IO a -> (IOError -> IO a) -> IO a
```

- Veatöötluspredikaate:

```
isEOFError, isUserError :: IOError -> Bool
isPermissionError       :: IOError -> Bool
isIllegalOperation      :: IOError -> Bool
isDoesNotExistError     :: IOError -> Bool
```

Madala taseme sisend/väljund

Näide:

```
module Main where
import IO

getAndOpenFile :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode
  = do putStr prompt
       name <- getLine
       catch (do handle <- openFile name mode
                 return handle)
             (\error -> do putStrLn ("Cannot open " ++ name)
                           print error
                           getAndOpenFile prompt mode)
```

Madala taseme sisend/väljund

```
main =
  do { fromHandle <- getAndOpenFile
                       "Copy from: " ReadMode
     ; toHandle   <- getAndOpenFile
                       "Copy to: "   WriteMode
     ; contents   <- hGetContents fromHandle
     ; hPutStr toHandle contents
     ; hClose fromHandle
     ; hClose toHandle
     ; putStrLn "Done."
     }
```

Kanalid ja paralleelsed protsessid

- Kanalid on fikseerimata mahutavusega puhvrid
- Kasutatakse paralleelprotsesside vahelises suhtlemises
- Operatsioonid kanalitega (teek `Channel`)

```
newChan          :: IO (Chan a)
writeChan        :: Chan a -> a -> IO ()
readChan         :: Chan a -> IO a
getChanContents  :: Chan a -> IO [a]
isEmptyChan      :: Chan a -> IO Bool
```

- Protsessi loomine (teek `ConcBase`)

```
forkIO :: IO () -> IO ()
```

Kanalid ja paralleelsed protsessid

- Näide:

```
module ChannelTest where

import Channel
import ConcBase

main :: IO ()
main = do c1 <- newChan :: IO (Chan Int)
          c2 <- newChan :: IO (Chan Int)
          forkIO (client c1 c2)
          forkIO (server c2 c1)
```

Kanalid ja paralleelsed protsessid

```
client :: Chan Int -> Chan Int -> IO ()
client cin cout = do writeChan cout 1
                     loop
     where loop = do c <- readChan cin
                     print c
                     writeChan cout c
                     loop

server :: Chan Int -> Chan Int -> IO ()
server cin cout  = do loop
      where loop = do c <- readChan cin
                      writeChan cout (c+1)
                      loop
```

Reaktiivsete animatsioonide kuvamine

- Reaktiivsete animatsioonide käivitamiseks tuleb siduda "tegelikud sündmused" (klahvi vajutused, . . .) meie abstraktsete sündmustega (tüüpi `Event`)
- Eesmärk on kirjutada funktsioon:

```
reactimate :: String -> Behavior a ->
                          (a -> IO Graphic) -> IO ()
```

- Meil on vaja "tekitada" striimide paar tüüpi

```
([Maybe UserAction],[Time])
```

- Kasutame funktsiooni:

```
windowUser :: Window ->
     IO (([Maybe UserAction],[Time]), IO ())
```

Reaktiivsete animatsioonide kuvamine

- Me kasutame funktsiooni `windowUser` kujul:

  ```
  ((us,ts),addEvents) <- windowUser w
  ```

- `us :: [Maybe UserAction]` ja `ts :: [Time]` on striimid, mis on “tekitatud” kasutades kanaleid
- `addEvents :: IO()` on aktsioon, mille käivitamisel lisatakse seni töötlemata kasutajategevused (“tegelikud sündmused”) ja nende ajatemplid striimidesse `us` ja `ts`

Reaktiivsete animatsioonide kuvamine

```
reactimate title franProg toGraphic = runGraphics $
  do w <- openWindowEx title (Just (0,0)) (Just (xWin,yWin))
          drawBufferedGraphic (Just 30)
     (user, addEvents) <- windowUser w
     addEvents
     let drawPic (Just p) = do g <- toGraphic p
                               setGraphic w g
                               addEvents
                               getWindowTick w
         drawPic Nothing  = return ()
     mapM_ drawPic
          (runEvent (sample `snapshot_` franProg) user)
```

Reaktiivsete animatsioonide kuvamine

```
sample :: Event ()
sample = Event (\(us,_) -> map aux us)
  where aux Nothing  = Just ()
        aux (Just _) = Nothing

runEvent :: Event a -> ([Maybe UserAction],[Time])
                    -> [Maybe a]
runEvent (Event fe) u = fe u

makeStream :: IO ([a], a -> IO ())
makeStream = do ch <- newChan
                contents <- getChanContents ch
                return (contents, writeChan ch)
```

Reaktiivsete animatsioonide kuvamine

```
windowUser w =
  do (evs, addEv) <- makeStream
     t0 <- timeGetTime
     let loop rt = do mev <- maybeGetWindowEvent w
                      case mev of
                        Nothing -> return ()
                        Just e  -> do addEv (Just e, rt)
                                      loop rt
     let addEvents = do t <- timeGetTime
                        let rt = w32ToTime (t-t0)
                        loop rt
                        addEv (Nothing, rt)
     return ((map fst evs, map snd evs), addEvents)
```