Exceptions:
Die Ausnahmen von der Regel

Heute wartet auf Sie:

- Letztes Mal: ein fehleranfälliger `talk`-Server
- Heute: Fehlerbehandlung
- Ausnahmebehandlung in Haskell98
- Unscharfe Ausnahmen
- Dynamische Ausnahmen
- Achtung, Ausnahmen!

Ausnahmebehandlung in Haskell98

Haskell 98: Fehler leben im IO-Monaden.

- Fehler fangen:

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

  - Variante: `try :: IO a -> IO (Either IOError a)`

- Fehler erzeugen:

```
userError :: String-> IOError
ioError   :: IOError-> IO a
```

Oder durch andere Operationen im IO-Monaden.

Fehler analysieren

Funktionen, die im Handler benutzt werden können:

```
isAlreadyExistsError   :: IOError -> Bool
isDoesNotExistError    :: IOError -> Bool
isAlreadyInUseError    :: IOError -> Bool
isFullError            :: IOError -> Bool
isEOFError             :: IOError -> Bool
isIllegalOperation     :: IOError -> Bool
isPermissionError      :: IOError -> Bool
isUserError            :: IOError -> Bool

ioeGetErrorString      :: IOError -> String
ioeGetHandle           :: IOError -> Maybe Handle
ioeGetFileName         :: IOError -> Maybe FilePath
```

Talk 0.2: Hauptprogramm

```
main :: IO ()
main =
  do port_num <- getArgs >>= return . read . head :: IO Integer
     s <- listenOn (PortNumber (fromInteger port_num))
     ch <- newChan
     loop s ch
```

- Kanäle nach Bedarf erzeugen

Talk 0.2: Hauptschleife

```
loop :: Socket -> Chan String -> IO ()
loop s ch =
  do (handle, wh, p) <- accept s
     hSetBuffering handle NoBuffering
     installHandler sigPIPE Ignore Nothing
     putStrLn ("New connection from "++ wh++
               " on port "++ show p)
     ch2 <- dupChan ch
     forkIO (catch (newUser handle wh ch2)
                   (\_ -> hClose handle))
     loop s ch2
```

- Fehlerbehandlung für `newUser`, `SIGPIPE` ignorieren

Talk 0.2: Benutzerprozess

```
newUser :: Handle-> String-> Chan String -> IO ()
newUser socket wh msgch =
  do hPutStrLn socket "Hello there. Please send your nickname."
     nick <- do nm <- hGetLine socket
                return (filter (not . isControl) nm)
     hPutStrLn socket ("Nice to meet you!")
     writeChan msgch (nick ++ "@" ++ wh ++ " has joined.")
```

Talk 0.2: Benutzerprozess (Forts.)

```
    wp <- forkIO write
    catch (read ((nick ++ ": ")++)) $
     \e-> do killThread wp
             if isEOFError e then
               writeChan msgch (nick++ "@"++ wh++ " has left.")
               else writeChan msgch (nick++ "@"++ wh++
                                     " left hastily ("++
                                     ioeGetErrorString e++ ")")
             hClose socket where
  read :: (String-> String)-> IO ()
  read f = hGetLine socket >>= writeChan msgch. f >> read f
  write :: IO ()
  write = readChan msgch >>= hPutStrLn socket >> write
```

Talk 0.2: Zusammenfassung

Nachteile:

- Kein Nachrichtenstau mehr
- Fehlerbehandlung für Benutzerprozess
- Anmeldeprozedur: Benutzer hat Namen
- Schnell verkaufen!

Probleme mit der Ausnahmebehandlung in Haskell98

- Keine Ausnahmebehandlung für rein funktionalen Code.
  - `error :: String-> a` bricht Programmausführung ab;
  - z.B. Fehler bei `read :: Read a=> String-> a`?
    - `readIO :: Read a=> String-> IO a` wirft Ausnahme
  - Laufzeitfehler (pattern match, fehlende Klassenmethoden, . . .)
- Keine Behandlung von asynchronen Ausnahmen möglich.
  - Nebenläufige Fehler, e.g. stack overflow, Speichermangel, Interrupts;

Probleme mit rein funktionalen Ausnahmen.

Warum nicht einfach `throw :: Exception-> a`?

- Wird die Ausnahme geworfen?

```
length [throw exception]
```

  Abhängig von Tiefe der Auswertung (wertet `length` Argument aus?)

- Welche Ausnahme wird geworfen:

```
throw ex1 + throw ex2
```

  Abhängig von Reihenfolge der Auswertung der Argumente

NB: Haskell98 ist spezifiert als nicht-strikt, aber nicht als *lazy*.

Auswertungsreihenfolge unspezifiziert!

Unscharfe Ausnahmen.

- Normale Ausnahmen:

  Wert eines Ausdrucks = Normaler Wert oder `Ausnahme`

```
data Maybe a  = Just a | Nothing
data Either a = Left String | Right a
```

- Unscharfe Ausnahmen:

  Wert eines Ausdrucks = Normaler Wert oder Menge von möglichen Ausnahmen

  ◦ Menge wird nicht konstruiert — semantisches Konstrukt.

Unscharfe Ausnahmen fangen.

- Ausnahmen fangen ist monadisch:
  - Funktion `bogus :: a-> (Exception-> a)-> a` hätte alten Probleme
- Determinisierung trennen von Ausnahmebehandlung:
  - `evaluate :: a-> IO a` wertet Ausdruck aus, wirft ggf. mögliche Ausnahme.
  - Ausnahme durch Auswertungsreihenfolge bestimmt.
- `catch :: IO a-> (Exception-> IO a)-> IO a` wie vorher.
- `Unscharfe` Ausnahmen können überall geworfen, aber nur im `IO`-Monaden gefangen werden.
  - Ausnahmen: semantisches `GOTO`.

Asynchrone Ausnahmen

Modelliert durch

```
throwTo :: ThreadId -> Exception-> IO ()
```

- Ausnahme wird in anderem Thread geworfen.
- Modelliert alle Situationen wie Interrupts etc.

Asynchrone Ausnahmen: Beispiel

Parallele Auswertung zweier IO-Statements:

- Wer zuerst fertig ist beendet Auswertung.

```
parIO :: IO a-> IO a-> IO a
parIO a1 a2 =
  do m <- newEmptyVar;
     c1 <- forkIO (a1 >>= putMVar m)
     c2 <- forkIO (a2 >>= putMVar m)
     r<- takeMVar m
     throwTo c1 Kill
     throwTo c2 Kill
     return r
```

Asynchrone Ausnahmen: Beispiel

Timeout-Operator:

- Wenn kein Ergbnis nach `n` Mikrosekunden, `Nothing`

```
timeout :: Int-> IO a-> IO (Maybe a)
timeout n a = parIO (r<- a; return (Just r))
                    (threadDelay n; return Nothing)
```

Unscharfe Ausnahmen: Benutzung

- Zur Benutzung: `import Control.Exception` (nur ghc)
- Achtung: per default normale Ausnahmen (Haskell98) definiert
  - Überlagerung durch Import, ggf. alte Funktionen verstecken:

```
import Prelude hiding (catch)
import IO hiding (catch,try)
```

- Ausnahmen fangen:

```
catch :: IO a-> (Exception -> IO a)-> IO a
try :: IO a -> IO (Either Exception a)
```

Dynamische Typen

Dynamische Typen (`Data.Dynamic`): Repräsentation aller Typen in einem Typen `Dynamic` zur Laufzieit.

```
class Typeable a where
  typeOf :: a -> TypeRep

toDyn :: (Typeable a) => a -> Dynamic
fromDyn :: (Typeable a) => Dynamic -> a -> a
fromDynamic :: (Typeable a) => Dynamic -> Maybe a
```

In einfachstem Fall (nicht-polymorph, first-order):

```
mkTyCon :: String -> TyCon
mkAppTy :: TyCon -> [TypeRep] -> TypeRep
```

Dynamische Ausnahmen: Beispiel

- Eigene Fehlerwerte definieren:

```
data MyError = Panic String
             | IOFailure Handle String
             | Warning Int String
```

- Als Instanz der Typklasse `Typeable`

```
instance Typeable MyError where
   typeOf _ = mkAppTy (mkTyCon "M.MyError") []
```

- Werfen: `throwDyn (Panic ‘‘HELP!!’’)`
- Fangen:

```
case e of DynException d->
  case fromDynamic d of Maybe (Panic str) -> ...
                        Maybe (IOFailure h str) -> ...
```

Ausnahmen: Achtung!

Ausnahmen und Nebenläufigkeit

- Ausnahmen können in anderen Thread geworfen werden!

```
ch <- newChan
forkIO (forever $ do readChan ch >>= putStrLn)
catch (do let x= ...
          writeChan ch x)
      (\e -> ...)
```

- Ausnahme wird in reader-Thread geworfen!
  - Abhilfe: Auswertung mit `evaluate` forcieren.

Ausnahmen: Achtung!

Fehlerabfrage ersetzt keine Ausnahmebehandlung:

```
b <- doesDirectoryExist name
when (not b) $ createDirectory name
```

- Zweite Aktion kann fehlschlagen!

Zusammenfassung

Heute Ausnahmezustand:

- Ausnahmen in Haskell98
- Unscharfe Ausnahmen
- Dynamische Ausnahmen

Nächste Woche: mein bester Freund ist eine C-Funktion, oder: das FFI.