```
                               Control.Parallel
                               ----------------
seq  :: a -> b -> b                    par  :: a -> b -> b
pseq :: a -> b -> b                    pseq a b = a ‘seq‘ lazy b

parMap f []     = []
parMap f (x:xs) = y ‘par‘ (ys ‘pseq‘ y:ys)
   where y = f x
         ys = parMap f xs

                          Control.Parallel.Strategies
                          ---------------------------
data Eval a = Done a   striktně vyhodnocovaná monáda
type Strategy a = a -> Eval a

using :: a -> Strategy a -> a
x ‘using‘ strat = runEval (strat x)

r0 :: Strategy a                       r0 x = return x
rseq :: Strategy a                     rseq x = x ‘pseq‘ return x
rdeepseq :: NFData a => Strategy a     rdeepseq x = rnf x ‘pseq‘ return x
rpar :: Strategy a                     rpar x = x ‘par‘ return x

parList :: Strategy a -> Strategy [a]
parList strat [] = return []
parList strat (x:xs) = do x’ <- rpar x
                          xs’ <- parList strat xs
                          return $ x’ : xs’

parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f = (‘using‘ parList strat) . map f

                      Dynamické typy v GHC, Data.Typeable
                      -----------------------------------
data TypeRep
data TypeCon
class Typeable a where
  typeOf :: a -> TypeRep
cast :: (Typeable a, Typeable b) => a -> Maybe b
gcast :: (Typeable a, Typeable b) => c a -> Maybe (c b)

mkTyCon       :: String -> TyCon
mkTyConApp    :: TyCon -> [TypeRep] -> TypeRep
mkAppTy       :: TypeRep -> TypeRep -> TypeRep
mkFunTy       :: TypeRep -> TypeRep -> TypeRep
splitTyConApp :: TypeRep -> (TyCon, [TypeRep])
funResultTy   :: TypeRep -> TypeRep -> Maybe TypeRep
typeRepTyCon  :: TypeRep -> TyCon
typeRepArgs   :: TypeRep -> [TypeRep]

class Show k => Key k
data SomeKey = forall k . Key k => SomeKey k
instance Key Bool
instance Key String
[ SomeKey True, SomeKey "cau" ]

toInt :: SomeKey -> Maybe Int
toInt (SomeKey k) = cast k

                    Výjimky v GHC, modul Control.Exception
                    --------------------------------------
data SomeException = forall e . Exception e => SomeException e

class (Typeable e, Show e) => Exception e where
  toException :: e -> SomeException
  fromException :: SomeException -> Maybe e
```

♣Funkce pro práci s výjimkami

```
throw :: Exception e => e -> a
throwIO :: Exception e => e -> IO a
catch :: Exception e => IO a -> (e -> IO a) -> IO a
catch (readFile f)
      (\e -> do let err = show (e :: IOException)
                hPutStr stderr ("Warning: Couldn’t open " ++ f ++ ": " ++ err)
                return "")

catchJust   :: Exception e => (e -> Maybe b) -> IO a -> (b -> IO a) -> IO a
handle      :: Exception e => (e -> IO a) -> IO a -> IO a
handleJust  :: Exception e => (e -> Maybe b) -> (b -> IO a) -> IO a -> IO a
try         :: Exception e => IO a -> IO (Either e a)
tryJust     :: Exception e => (e -> Maybe b) -> IO a -> IO (Either b a)

finally     :: IO a -> IO b -> IO a
onException :: IO a -> IO b -> IO a
bracket     :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
withFile name mode = bracket (openFile name mode) hClose
assert      :: Bool -> a -> a
```

♣Standardní hierarchie výjimek

```
data IOException
data ArithException = Overflow|Underflow|LossOfPrecision|DivideByZero|Denormal
data ArrayException = IndexOutOfBounds String | UndefinedElement String
data AssertionFailed = AssertionFailed String
data AsyncException = StackOverflow | HeapOverflow | ThreadKilled |UserInterrupt
data NonTermination = NonTermination
data NestedAtomically = NestedAtomically
data BlockedOnDeadMVar = BlockedOnDeadMVar
data BlockedIndefinitely = BlockedIndefinitely
data Deadlock = Deadlock
data PatternMatchFail = PatternMatchFail String
data RecConError = RecConError String
data RecSelError = RecSelError String
data RecUpdError = RecUpdError String
data ErrorCall = ErrorCall String
```

♣Uživatelsky definovaná jednoduchá výjimka

```
data MyException = ThisException | ThatException deriving (Show, Typeable)
instance Exception MyException
```

♣Uživatelsky definovaná hierarchie vyjímek

Předek všech výjimek v kompilátoru.

```
data SomeCompilerException = forall e . Exception e => SomeCompilerException e
  deriving Typeable
instance Show SomeCompilerException where show (SomeCompilerException e) =show e
instance Exception SomeCompilerException

compilerExceptionToException :: Exception e => e -> SomeException
compilerExceptionToException = toException . SomeCompilerException
compilerExceptionFromException :: Exception e => SomeException -> Maybe e
compilerExceptionFromException x = do SomeCompilerException a <- fromException x
                                      cast a
```

Podtřída `SomeCompilerException`, ale také ještě předek dalších výjimek.

```
data SomeFrontendException = forall e . Exception e => SomeFrontendException e
    deriving Typeable
instance Show SomeFrontendException where show (SomeFrontendException e) =show e
instance Exception SomeFrontendException where
    toException = compilerExceptionToException
    fromException = compilerExceptionFromException
frontendExceptionToException = toException . SomeFrontendException
frontendExceptionFromException x = do SomeFrontendException a <- fromException x
                                      cast a
```

Konkrétní výjimka, potomek `SomeFrontendException`.

```
data MismatchedParentheses = MismatchedParentheses deriving (Typeable, Show)
instance Exception MismatchedParentheses where
    toException   = frontendExceptionToException
    fromException = frontendExceptionFromException
```

## Vícevláknové programování v GHC, Control.Concurrency

```
data ThreadId
myThreadId  :: IO ThreadId
forkIO      :: IO () -> IO ThreadId
killThread  :: ThreadId -> IO ()
throwTo     :: Exception e => ThreadId -> e -> IO ()
yield       :: IO ()
threadDelay :: Int -> IO ()

mergeIO   :: [a] -> [a] -> IO [a]
nmergeIO  :: [[a]] -> IO [a]

test1 m = do threadDelay 3000000          w1 = do m1 <-newEmptyMVar
             putStrLn "b"                         m2 <-newEmptyMVar
             putMVar m ()                         m3 <-newEmptyMVar
test2 m = do threadDelay 1000000                  forkIO (test1 m1)
             putStrLn "a"                         forkIO (test2 m2)
             putMVar m ()                         forkIO (test3 m3)
test3 m = do threadDelay 5000000                  takeMVar m1
             putStrLn "c"                         takeMVar m2
             putMVar m ()                         takeMVar m3
```

Při kompilaci s `-rtsopts -threaded` lze program spustit pomocí *+RTS -N*, případně *+RTS -N8* a Haskell bude používat více systémových vláken.

Problém: synchronizace proměnných mezi vlákny:

```
inc :: IORef Int -> IO ()
inc ref = do v <- readIORef ref
             writeIORef ref (v + 1)

main = do ref <- newIORef 0
          sequence_ (replicate 100000 $ forkIO $ inc ref)
          threadDelay 1000
          v <- readIORef ref
          print v
```

Funkce `main` vypisuje: `94921 91195 94900 91211`

```
module Control.Concurrent.MVar
  data MVar a
  newMVar :: a -> IO (MVar a)
  newEmptyMVar :: IO (MVar a)
  takeMVar :: MVar a -> IO a
  putMVar :: MVar a -> a -> IO ()

  readMVar :: MVar a -> IO a
  swapMVar :: MVar a -> a -> IO a

  tryTakeMVar :: MVar a -> IO (Maybe a)
  tryPutMVar :: MVar a -> a -> IO Bool
  isEmptyMVar :: MVar a -> IO Bool

  withMVar :: MVar a -> (a -> IO b) -> IO b
  modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
  modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b

inc :: MVar Int -> IO ()
inc var = do v <- takeMVar var
             putMVar var (v + 1)

main = do var <- newMVar 0
          sequence_ (replicate 100000 $ forkIO $ inc var)
          threadDelay 1000
          v <- takeMVar var
          print v
```

♣Ukončování programu −− jakmile skončí `main`, skončí všechno. Můžeme tomu samozřejmě zabránit:

```
{-# NOINLINE children #-}
children :: MVar [MVar ()]
children = unsafePerformIO (newMVar [])

waitForChildren = takeMVar children >>= mapM_ takeMVar

forkChild :: IO () -> IO ThreadId
forkChild io = do mvar <- newEmptyMVar
                  mvars <- takeMVar children
                  putMVar children (mvar:mvars)
                  forkIO (io ‘finally‘ putMVar mvar ())
main = do ...
          waitForChildren

module Control.Concurrent.QSem         module Control.Concurrent.QSemN
  data QSem                              data QSemN
  newQSem :: Int -> IO QSem              newQSemN :: Int -> IO QSemN
  waitQSem :: QSem -> IO ()              waitQSemN :: QSemN -> Int -> IO ()
  signalQSem :: QSem -> IO ()            signalQSemN :: QSemN -> Int -> IO ()

module Control.Concurrent.Chan
  data Chan a
  newChan :: IO (Chan a)
  writeChan :: Chan a -> a -> IO ()
  readChan :: Chan a -> IO a

  dupChan :: Chan a -> IO (Chan a)
  unGetChan :: Chan a -> a -> IO ()

  isEmptyChan :: Chan a -> IO Bool
```

Implementace

```
  type Stream a = MVar (ChItem a)
  data ChItem a = ChItem a (Stream a)
  data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
  newChan = do hole  <- newEmptyMVar
               readVar  <- newMVar hole
               writeVar <- newMVar hole
               return (Chan readVar writeVar)
  writeChan (Chan _ writeVar) val = do new_hole <- newEmptyMVar
                                       modifyMVar_ writeVar $ \old_hole -> do
                                         putMVar old_hole (ChItem val new_hole)
                                         return new_hole
  readChan (Chan readVar _) = do modifyMVar readVar $ \read_end -> do
                                  (ChItem val new_read_end) <- readMVar read_end
                    -- Use readMVar here, not takeMVar, else dupChan doesn’t work
                                  return (new_read_end, val)
  dupChan (Chan _ writeVar) = do hole       <- readMVar writeVar
                                 newReadVar <- newMVar hole
                                 return (Chan newReadVar writeVar)
```

## Network a sockety, high-level interface

♣Klient:

```
import Network; import System.IO
main = withSocketsDo $ do
  handle <- connectTo "atrey" (PortNumber 13)
  hSetNewlineMode handle universalNewlineMode
  str <- hGetContents handle
  putStr str
```

♣Server:

```
import Control.Monad; import Control.Concurrent; import Network;import System.IO
main = withSocketsDo $ do socket <- listenOn (PortNumber 13)
                          forever (accept socket >>= forkIO . respond)
 where respond (handle, hostname, port) = do
         hPutStrLn handle $ "Connection from " ++ hostname ++ ":" ++ show port
         hClose handle
```