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

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

                     Control.Parallel.Strategies
                     ---------------------------
type Done = ()
done = ()

type Strategy a = a -> Done
rwhnf :: Strategy a
rwhnf x = x `pseq` done
class NFData a where rnf :: Strategy a

parList :: Strategy a -> Strategy [a]
parList strat [] = done
parList strat (x:xs) = strat x `par` parList strat xs

using :: a -> Strategy a -> a
using x s = s x `pseq` x

psum :: [Int] -> Int
psum xs = sum xs `using` parList rwhnf

                  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

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
```

♣ Jednoduchá výjimka

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

♣ Hierarchie vyjímek

```
-- Make the root exception type for all the exceptions in a compiler
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
-- Make a subhierarchy for exceptions in the frontend of the compiler
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 :: Exception e => e -> SomeException
frontendExceptionToException = toException . SomeFrontendException
frontendExceptionFromException :: Exception e => SomeException -> Maybe e
frontendExceptionFromException x = do
    SomeFrontendException a <- fromException x
    cast a

-- Make an exception type for a particular frontend compiler exception
data MismatchedParentheses = MismatchedParentheses deriving (Typeable, Show)
instance Exception MismatchedParentheses where
    toException   = frontendExceptionToException
    fromException = frontendExceptionFromException
```

♣ Funkce pro práci s výjimkami

```
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
```

```
              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]

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

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)
```