Úkol z minula – acc

```
import ...
acc f xs =
   do let len=length xs
      vars<-replicateM len newEmptyMVar
      let a=(listArray (0,len)) vars
          compute m i x | i.&.m/=0 || i+m>=len = putMVar (a!i) x
                        | otherwise = takeMVar (a!(i+m)) >>= compute (m+m) i.f x
      mapM_ (forkIO.uncurry (compute 1)) (zip [0..] xs) >> takeMVar (a!0)
```

Control.Concurrent.QSem

```
data QSem
newQSem   :: Int -> IO QSem
waitQSem  :: QSem -> IO ()
signalQSem:: QSem -> IO ()
```

Střeva *QSem*u

```
newtype QSem = QSem (MVar (Int, [MVar ()]))
newQSem init = do sem <- newMVar (init,[])
                  return (QSem sem)
waitQSem (QSem sem) = do (avail,blocked) <- takeMVar sem  -- gain ex. access
                         if avail > 0 then putMVar sem (avail-1,[])
                          else do block <- newEmptyMVar
                                  putMVar sem (0, blocked++[block])
                                  takeMVar block
signalQSem (QSem sem) = do (avail,blocked) <- takeMVar sem
                           case blocked of [] -> putMVar sem (avail+1,[])
                                           (block:blocked') -> do
                                             putMVar sem (0,blocked')
                                             putMVar block ()
```

Control.Parallel revisited, Control.Parallel.Strategies

```
par :: a -> b -> b
pseq:: a -> b -> b

type Done = ()
type Strategy a = a -> Done
r0 :: Strategy a                          r0 x = ()-}
rwhnf :: Strategy a                       rwhnf x = x `seq` ()-}
class NFData a where rnf :: Strategy a    default je rnf=rwhnf

(>|) :: Done -> Done -> Done              seq
(>||) :: Done -> Done -> Done             par
demanding :: a -> Done -> a               flip seq
sparking ::  a -> Done -> a               flip par

($|) :: (a->b)->Strategy a->a->b          f $| s  = \ x -> f x `demanding` s x
($||):: (a->b)->Strategy a->a->b          f $|| s = \ x -> f x `sparking` s x
(.|) :: (b->c)->Strategy b->(a->b)->a->c
(.|)  f s g = \ x -> let gx = g x in f gx `demanding` s gx
(.||):: (b->c)->Strategy b->(a->b)->a->c
(.||) f s g = \ x -> let gx = g x in f gx `sparking` s gx

(seq/par)Pair   ::Strategy a->Strategy b->Strategy (a, b)
(seq/par)Triple ::Strategy a->Strategy b->Strategy c->Strategy (a, b, c)
(seq/par)List   ::Strategy a->Strategy [a]
(seq/par)ListN  ::Integral b=>b->Strategy a->Strategy [a]
(seq/par)ListNth::Int->Strategy a->Strategy [a]
parMap          ::Strategy  b -> (a-> b) ->[a]->[b]
parFlatMap      ::Strategy [b]-> (a->[b])->[a]->[b]
parZipWith      ::Strategy c->(a->b->c)->[a]->[b]->[c]
```

Synchronizace pomoci Control.Monad.STM

```
inc x s = do v<-readIORef x
             writeIORef x (v + 1)
             signalQSem s
doInc n r = do s <- newQSem 0
               sequence_ (replicate n $ forkIO (inc r s))
               sequence_ (replicate n $ waitQSem s)
w2 = do r <- newIORef 0
        doInc 100000 r
        readIORef r
```

Vraci hodnoty jako 99886, 99908,...

```
data STM a                              atomically:: STM a -> IO a
retry    :: STM a                       orElse    :: STM a -> STM a -> STM a

data TVar a                             newTVar  :: a -> STM (TVar a)
readTVar :: TVar a -> STM a             writeTVar:: TVar a -> a -> STM ()
incS x s = do atomically $ do v <- readTVar x
                              writeTVar x (v + 1)
              signalQSem s
doIncS n r = do s <- newQSem 0
                sequence_ (replicate n $ forkIO (incS r s))
                sequence_ (replicate n $ waitQSem s)
w3 = do r <- atomically (newTVar 0)
        doIncS 100000 r
        atomically (readTVar r)
```

Synchronizace

Čekání na událost

```
produce::TVar [Int]->Int->IO ()
produce q n = do atomically $ do s <- readTVar q
                                 writeTVar q (n : s)
consume::TVar Int->TVar [Int]->IO ()
consume e q = do s <- atomically $ do l <- readTVar q
                                      if length l < 100 then retry
                                          else do writeTVar q (drop 100 l)
                                                  return (take 100 l)
                 putStrLn (show $ sum s)
                 atomically $ do k <- readTVar e
                                 writeTVar e (k-1)
w4 = do s <- atomically (newTVar [])
        e <- atomically (newTVar 10)
        sequence_ (replicate 10 $ (forkIO $ consume e s))
        mapM_ (\n -> forkIO (produce s n)) [1..1000]
        atomically $ do ne <- readTVar e
                        if ne /= 0 then retry else return ()
```

Template Haskell

```
sum 1 = [| id |]
sum n = [| \a -> $(sum (n-1)).(+a) |]
```

V `ghc` nebo `ghci` je třeba svičík `-XTemplateHaskell` (ve starších verzích `-fth`).
Potom v `ghci` je `:t $(sum 3)` typu `(Num a) => a -> a -> a -> a`. `:t sum` je `(Num t)=>t->ExpQ`.

```
printf str = p str [| [] |] where
  p [] a = a
  p ('%':'s':ss) a = [| \s -> $(p ss [| $a ++       s  |]) |]
  p ('%':'d':ss) a = [| \d -> $(p ss [| $a ++ (show d) |]) |]
  p (c:ss) a = p ss  [| $a ++ [c] |]
```

Potom `:t $(printf "%s ma %d psu")` je `(Show a) => [Char] -> a -> [Char]`
Typ `[| ... |]` je `ExpQ`

```
type ExpQ = Q Exp; data Exp = VarE Name | ConE Name | LitE Lit | AppE Exp Exp |
  InfixE (Maybe Exp) Exp (Maybe Exp) | LamE [Pat] Exp | TupE [Exp] |
  CondE Exp Exp Exp | LetE [Dec] Exp | CaseE Exp [Match] | DoE [Stmt] | ...
sel i n = [| \x -> $(caseE [|x|] [alt]) |] where
  alt = match pat rhs []
  pat = tupP (map varP as)
  rhs = normalB (varE (as !! (i-1)))
  as = [mkName ("a"++show i) | i<-[1..n]]
sel2 n m = do x<-newName "x"
              lam1E (tupP (replaceAt n (replicate m wildP) (varP x))) (varE x)
replaceAt n xs x  =  take (n-1) xs ++ x : drop n xs
```