```
                              Control.Monad.STM
                              -----------------
data STM a

atomically:: STM a -> IO a
retry     :: STM a
orElse    :: STM a -> STM a -> STM a
catchSTM  :: STM a -> (Exception -> STM a) -> STM a

data TVar a
newTVar  :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar:: TVar a -> a -> STM ()
```

♣ Synchronizace

```
incS::TVar Int->QSem->IO ()
incS x s = do atomically $ do v <- readTVar x
                              writeTVar x (v + 1)
                              signalQSem s 1
doIncS::Int -> TVar Int -> IO ()
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
        x <- atomically (readTVar r)
        putStrLn (show x)
```

♣ Finanční transakce

```
transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = atomically $ do withdraw from amount
                                          deposit to amount
type Account = TVar Int

withdraw :: Account -> Int -> STM ()
withdraw from amount = do bal <- readTVar from
                          writeTVar from (bal - amount)

limitedWithdraw from amount = do bal <- readTVar from
                                 if amount > 0 && amount > bal
                                  then retry
                                  else writeTVar from (bal - amount)
```

♣ 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 ()
```

GADT, zobecněné algebraické typy

Rozšíření, lze zapnout pomocí −*XGADTs*

```
data Term a where Lit    :: Int -> Term Int
                  Succ   :: Term Int -> Term Int
                  IsZero :: Term Int -> Term Bool
                  If     :: Term Bool -> Term a -> Term a -> Term a
                  Pair   :: Term a -> Term b -> Term (a,b)
```

`eval :: Term a -> a` Ta typová signatura `evalu` je důležitá, bez ní to nefunguje

```
eval (Lit i)       = i
eval (Succ t)      = 1 + eval t
eval (IsZero t)    = eval t == 0
eval (If b e1 e2) = if eval b then eval e1 else eval e2
eval (Pair e1 e2) = (eval e1, eval e2)
```

Funguje dokonce i

```
eval :: Term a -> a -> a
eval (Lit i) j =  i+j
```

When pattern−matching against data constructors drawn from a GADT, for example in a case expression, the following rules apply:

- the type of the scrutinee must be rigid.
- the type of the entire case expression must be rigid
- the type of any free variable mentioned in any of the case alternatives must be rigid.

A type is rigid if it is completely known to the compiler at its binding site. The easiest way to ensure that a variable has a rigid type is to give it type signature. For more precise details see Simple unification−based type inference for GADT.

SafeListy

```
data Empty
data NonEmpty

data SafeList x y where Nil :: SafeList x Empty
                        Cons:: x -> SafeList x y  -> SafeList x NonEmpty
safeHead :: SafeList x NonEmpty -> x
safeHead (Cons x _) = x
```

Ale co funkce

```
silly 0       = Nil
silly n | n>0 = Cons n $ silly (n−1)
```

Můžeme uvolnit podmínky na `Cons` a získat

```
data NotSafe
data Safe

data MarkedList t u where Nil  :: MarkedList t NotSafe
                          Cons :: t -> MarkedList t y -> MarkedList t z
safeHead                    ::  MarkedList x Safe -> x
safeHead (Cons x _)          =   x

silly 0       = Nil
silly n | n>0 = Cons () $ silly (n−1)
```

GADT zobecňují existenciální datové typy

```
data Foo = forall a. MkFoo a (a -> Bool)
data Foo' where MKFoo :: a -> (a->Bool) -> Foo'
```

GADT s kontexty u parametrů zpřístupňují po pattern−matchingu příslušné dictionaries

```
data Set a where  MkSet :: Eq a => [a] -> Set a
```

Je tu rozdíl oproti Haskell 98 deklaraci `data Eq a => Set' a = MkSet' [a]`!

```
makeSet :: Eq a => [a] -> Set a
makeSet xs = MkSet (nub xs)

insert :: a -> Set a -> Set a
insert a (MkSet as) | a `elem` as = MkSet as
                    | otherwise   = MkSet (a:as)
```

Mohou poskytovat explicitní dictionaries

```
data NumInst a where MkNumInst :: Num a => NumInst a

intInst :: NumInst Int
intInst = MkNumInst

plus :: NumInst a -> a -> a -> a
plus MkNumInst p q = p + q
```

Rychlé Stringy

String má velký overhead –– je to (líný) seznam odkazů na 32bitové chary
Existuje proto typ `Data.ByteString`, nad kterým jsou definovány klasické operace pro `Word8`

```
empty singleton pack unpack cons snoc append head uncons last tail init null
length map reverse intersperse intercalate transpose foldl foldl1 foldr foldr1
concat concatMap any all maximum minimum scanl scanl1 scanr scanr1 mapAccumL
mapAccumR mapIndexed replicate unfoldr unfoldrN take drop splitAt takeWhile
dropWhile span spanEnd break breakEnd group groupBy inits tails split splitWith
isPrefixOf isSuffixOf isInfixOf isSubstringOf findSubstring findSubstrings elem
notElem find filter partition index elemIndex elemIndices elemIndexEnd findIndex
findIndices count zip zipWith unzip sort copy packCString packCStringLen
useAsCString useAsCStringLen getLine getContents putStr putStrLn interact
readFile writeFile appendFile hGetLine hGetContents hGet hGetNonBlocking hPut
hPutStr hPutStrLn
```

Typ `Data.ByteString.Char8` má metody s `Char`em místo `Word8`, z `Char`ů se používá jenom 8 bitů.
Oba tyto typy mají celý string v jednom kusu paměti –– existují i líné varianty
`Data.ByteString.Lazy` a `Data.ByteString.Lazy.Char8`, které pracují s chunky po 64k.

Implementace `Data.ByteString` je viditelná v `Data.ByteString.Internal`:

```
data ByteString = PS !(ForeignPtr Word8) !Int !Int
fromForeignPtr :: ForeignPtr Word8 -> Int -> Int -> ByteString
toForeignPtr :: ByteString -> (ForeignPtr Word8, Int, Int)
w2c :: Word8 -> Char
c2w :: Char -> Word8
```

Přetížené řetězcové literály

Nutno zapnout rozšíření pomocí `-XOverloadedStrings`

```
class IsString a where fromString :: String -> a
```

Jediná defaultní instance je

```
instance IsString [Char] where fromString = id
```

ale každý `ByteString` definuje svojí vlastní instanci

Regulární výrazy

Obecný interface pro libovolný regexpový engine `Text.Regex.Base`

```
class Extract source where
  before :: Int->source->source        after :: Int -> source -> source
  empty :: source                      extract :: (Int, Int) -> source -> source
instance Extract String, instance Extract ByteString

class Extract s => RegexLike r s where
  matchAll::r->s->[MatchArray]          matchAllText::r->s->[MatchTexts]
  matchOnce::r->s->MaybeMatchArray      matchOnceText::r->s->Maybe(s,MatchTexts,s)
  atchCount::r->s->Int                 matchTest::r->s->Bool

class RegexLike r s => RegexContext r s target where
  match :: r -> s -> target             matchM :: Monad m => r -> s -> m target
```

Existují instance typu `RegexLike a b => RegexContext a b Neco`, kde `Neco` je:

```
Bool Int ()
(MatchOffset, MatchLength) (b,b,b), (b,b,b,[b]), (b, MatchText b, b)
MatchArray [(MatchOffset,MatchLength)] [b] (MatchResult b) (Array Int b)
[Array Int b] [MatchArray] [MatchText b] [[b]]
type MatchOffset = Int                      type MatchLength = Int
type MatchArray = Array Int (MatchOffset, MatchLength)
type MatchText source = Array Int (source, (MatchOffset, MatchLength))
```

```
(=~)::(RegexMaker Regex ... s, RegexContext Regex s1 t)=>s1->s->t
(=~~)::(RegexMaker Regex ... s, RegexContext Regex s1 t, Monad m)=>s1->s->m t
```

Grep je pak

```
main = do a<-getArgs
          case a of [r]      ->grep r
                    otherwise->putStrLn "Usage: grep regex"
  where grep r = interact $ unlines . (filter (=~r)) . lines
```

Implementace

```
Backend         Grouping?       POSIX/Perl      Speed           Native Impl?
----------------------------------------------------------------------------
regex-posix     Yes             POSIX           very slow       No
regex-parsec    Yes             POSIX,Perl      slow            Yes
regex-tre       Yes             POSIX           fast            No
regex-tdfa      Yes             POSIX           fast            Yes
regex-pcre      Yes             Perl            fast            No
regex-dfa       No              POSIX           fast            Yes
```