```
                          Z A K L A D N I    D E F I N I C E
                          -----------------------------------
--Co je to arrow
class Arrow a where
    --povinne
    arr   :: (b->c) -> a b c
    (>>>) :: a b c -> a c d -> a b d
    first :: a b c -> a (b,d) (c,d)

    --nepovinne, lze definovat z povinnych
    second :: a b c -> a (d,b) (d,c)
    second f = arr swap >>> first f >>> arr swap
     where
      swap (x,y) = (y,x)

    (***) :: a b1 c1 -> a b2 c2 -> a (b1,b2) (c1,c2)
    f *** g = first f >>> second g

    (&&&) :: a b c1 -> a b c2 -> a b (c1,c2)
    f &&& g = arr (\b->(b,b)) >> (f *** g)

--Monady jsou Arrow
newtype Kleisli m a b = K (a -> mb)
instance Monad m => Arrow (Kleisli m) where
    arr f = K (\b -> return (f b))                              -- K (return $ f id)
    K f >>> K g = K (\b -> f b >>= g)                           -- K (f id >>= g)
    first (K f) = K (\(b,d) -> f b >>= \c -> return (c,d))

--Dalsi rozsireni Arrow
class Arrow a => ArrowZero a where zeroArrow :: a b c
class Arrow a => ArrowPlus a where (+++) :: a b c -> a b c -> a b c

--Choice
class Arrow a => ArrowChoice a where
    --povinne
    left :: a b c -> a (Either b d) (Either c d)

    --nepovinne
    right f = arr mirror >>> left f >>> arr mirror
     where
      mirror (Left x) = Right x
      mirror (Right x) = Left x

    f <+> g = left f >>> right g

    f ||| g = (f <+> g) >>> arr untag
     where
      untag (Left x) = x
      untag (Right x) = x

instance Monad m => ArrowChoice (Kleisli m) where
    left (K f) = K (\x -> case x of Left b -> f b >>= \c -> return (Left c)
                                    Right d -> return (Right d))

--Apply
class Arrow a => ArrowApply a where
    app :: a (a b c , b) c

instance Monad m => ArrowApply (Kleisli m) where
    app = K( \ (K f, x) -> f x)


--ArrowApply uz je Monada
newtype ArrowApply a => ArrowMonad a b = M (a Void b)
instance ArrowApply a => Monad (ArrowMonad a) where
    return x = M (arr (\z -> x))
    M m >>= f = M ( m >>>
                    arr (\x -> let M h = f x in (h, ())) >>>
                    app )
```

S T A T E   A R R O W

```
prod::(a1->b1)->(a2->b2)->(a1,a2)->(b1,b2)
(f `prod` g) (a1,a1) = (f a1, g a2)

newtype State s a b = ST {unST :: ((s,a)->(s,b))}
instance Arrow (State s) where
    arr f = ST (id `prod` f)
    ST f >>> ST g = ST (g . f)
    first (ST s) = ST (assoc . (f`prod`id) . unassoc) where
       assoc   ((a,b),c) = (a,(b,c))
       unassoc (a,(b,c)) = ((a,b),c)

fetch::State s () s
fetch = ST (\(s,_) -> (s,s))

store::State s s ()
store = ST (\(_,s')->(s',()))

nextNum::State Int () Int
nextNum = fetch >>> arr ((+1)`prod`id) >> first store >>> arr snd
```

I N T E R P R E T E R   V   M O N A D A C H   A   A R R O W

```
--1) promenne a cisla
data Exp = Var String | Add Exp Exp
data Val = Cislo Int
type Env = [(String, Val)]

--M O N A D Y                                A R R O W
eval::Exp -> Env -> M Val                    eval::Exp -> A Env Val
eval (Var s) env = return (lookup s env)     eval (Var s) = arr (lookup s)
eval (Add e1 e2) env =                       eval (Add e1 e2) =
  liftM2 add (eval e1 env) (eval e2 env)       liftA2 add (eval e1) (eval e2)
   where                                     --((eval e1) &&& (eval e2)) >>>
    add (Cislo a) (Cislo b) = Cislo (a+b)    --arr (\(a,b) -> a 'add b)

--2) podminky
data Exp = ... | If Exp Exp Exp
data Val = ... | Bl Bool

--M O N A D Y                         A R R O W
eval (If e1 e2 e3) env = do           eval (If e1 e2 e3)=(eval e1 &&& arr id)>>>
  podm <- eval e1 env     arr(\(Bl b,env)->if b then Left env else Right env)>>>
  if b then eval e2 env                                     (eval e2 ||| eval e3)
  else eval e3 env

--3) lambda-kalkulus
data Exp = ...|Lam String Exp | App Exp Exp
data Val = ...|Fun (Val -> M Val) -- monady
data Val = ...|Fun (A Val Val) -- arrow

--M O N A D Y                                A R R O W
eval (Lam x e) env =                         eval (Lam x e) = arr (\env ->
  return (Fun (\v -> eval e((x,v) : env))      Fun(arr(\v->(x,v):env)>>>eval e))
eval (App e1 e2) env=eval e1 env>>=          eval (App e1 e2) =
  \Fun f -> eval e2 env >>= \v -> f v   ((eval e1>>>arr(\Fun f->f)) &&& eval e2)
                                                                         >>> app
--DO NOTACE
eval (Add e1 e2) env = do                    eval (Add e1 e2) = proc env -> do
  Cislo a1 <- eval e1 env                      Cislo a1 <- eval e1 -< env
  Cislo a2 <- eval e2 env                      Cislo a2 <- eval e2 -< env
  return $ Cislo (a1 + a2)                     returnA $ Cislo (a1 + a2)

eval (If e1 e2 e3) env = do                  eval (If e1 e2 e3) = proc env -> do
  podm <- eval e1 env                          podm <- eval e1 -< env
  if b then eval e2 env else eval e2 env       (eval e2 ||| eval e2) -<
                                                         if podm then Left env
                                                                  else Right env
```

```haskell
nextNum::State Int () Int
nextNum = proc () -> do
            n <- fetch -< ()
            store -< (n+1)
            returnA -< n

                       S T R E A M   P R O C E S O R Y
                       -------------------------------
data SP a b = Put b (SP a b) | Get (a -> SP a b)

instance Arrow SP where
    arr f = Get (\x->Put (f x) (arr f))

    sp1       >>> Put c sp2 = Put c (sp1>>>sp2)
    Put b sp1 >>> Get f     = sp1 >>> f b
    Get f1    >>> sp2       = Get (\x -> f1 x >>> sp2)

    first f = bypass [] f where
      bypass ds (Get f) = Get (\(b,d)->bypass (ds++[d]) (f b))
      bypass (d:ds) (Put c sp) = Put (c,d) (bypass ds sp)
      bypass [] (Put c sp) = Get (\(b,d)->Put (c,d) (bypass [] sp))

fibs::SP Int Int
fibs = Put 0 fibs' where fibs' = Put 1 (liftA2 (+) fibs fibs')


instance ArrowZero SP where zeroArrow = Get (\x->zeroArrow)


instance ArrowPlus SP where Put b sp1 <+> sp2 = Put b (sp1<+>sp2)
                            sp1 <+> Put b sp2 = Put b (sp1<+>sp2)
                            Get f1 <+> Get f2 = Get (\a->f1 a<+>f2 a)


instance ArrowChoice SP where left (Put c sp)=Put (Left c) (left sp)
                              left (Get f) = Get (\z->case z of
                                     Left a->left (f a)
                                     Right a->Put (Right a) (left (Get f)))



         P A R S E R   S E   S T A T I C K O U   I N F O R M A C I
         ---------------------------------------------------------
data StaticParser      = SP Bool [Char]
data DynamisParser a b = DP ((a,String)->(b,String))
data Parser        a b = P StaticParser (DynamicParser a b)

instance Arrow Parser where
    arr f = P (SP True []) (DP (\(b,s)->(f b,s)))

    P (SP e1 s1) (DP p1) >>> P (SP e2 s2) (DP p2) =
        P (SP (e1 && e2) (s1 `union` if e1 then s2 else []))
          (DP (p2 . p1))


    first (P sp (DP p)) = P sp (\((b,d),s)->let (c,s')=p (b,s) in ((c,d),s')

instance ArrowZero Parser where zeroArrow = P (SP False []) (DP undefined)
instance ArrowPlus Parser where P (SP e1 s1) (DP p1) <+> P (SP e2 s2) (DP p2) =
   P (SP (e1 || e2) (s1 ++ s2)) (DP (\(b,s)->case s of
                                  []  -> if e1 then p1 [] else p2 []
                                  c:cs-> if c`elem`s1 then p1 s else
                                         if c`elem`s2 then p2 s else
                                         if e1 then p1 s else p2 s))
--neni ArrowChoice ani ArrowApply
```

```
                 N E D E T E R M I N I S T I C K E   V Y P O C T Y
                 -------------------------------------------------
newtype NonDet a b = ND (a->[b])

instance Arrow NonDet where
    arr f = ND (\a->[f a])
    ND f >>> ND g = ND (\a->[c | b<-f a,c<-g b])
    first (ND f) = ND (\(b,d)->[(c,d) | c<-f b])

instance ArrowZero NonDet where zeroArrow = ND (\_->[])
instance ArrowPlus NonDet where ND f <+> ND g = ND (\a->f a ++ g a)

instance ArrowChoice NonDet where
    left (ND f) = ND (\i->case i of Left a ->then Left (f b)
                                    Right b->then Right [b])

instance ArrowApply NonDet where
    app = ND (\(ND f,a) -> f a)



                             Z M E N Y   C H O V A N I
                             -------------------------
newtype MapTrans t a b = MT ((t->a) -> (t->b))

instance Arrow (MapTrans t) where
    arr f = ND (f .)
    MT f >>> MF g = MT (g . f)
    first (MT f) = MT (zipMap . (f`prod`id) . unzipMap) where
       zipMap   ::(t->a,t->b)->(t->(a,b))
       zipMap   (f,g) t = (f t, g t)
       unzipMap::(t->(a,b))->(t->a,t->b)
       unzipMap fg = (fst fg, snd fg)

--neni ArrowChoice
--neni ArrowApply
```