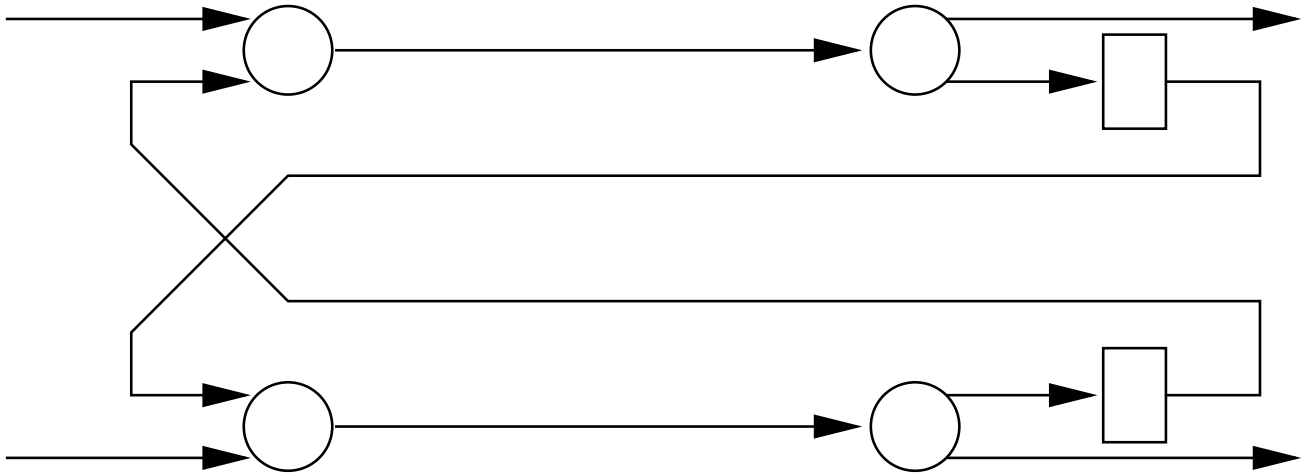
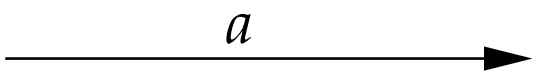
Skeemid

- Eesmärk: esitada riistvara skeeme ja teisi andmevoodiagrammidel baseeruvaid kirjeldusi Haskellis

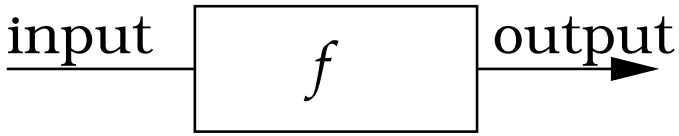


Skeemid

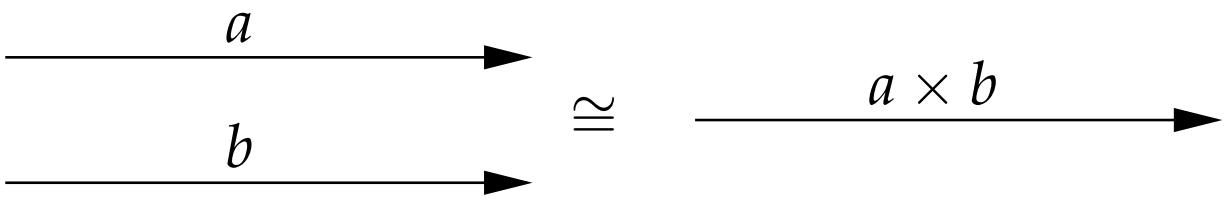
- Skeemid koosnevad juhtmetest ja komponentidest
- Läbi juhtmete ”voolavad” etteantud tüüpi väärtused



- Komponendid omavad sisend- ja väljundjuhtmeid



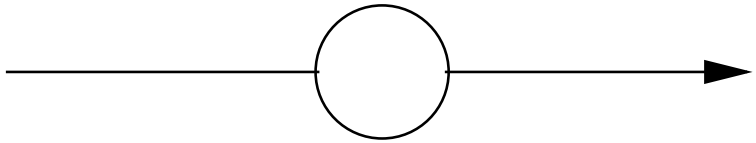
- Kommunikatsioon on sünkroone



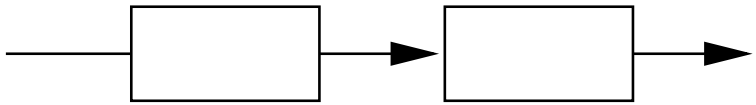
Skeemid

- Skeemide konstrueerimine

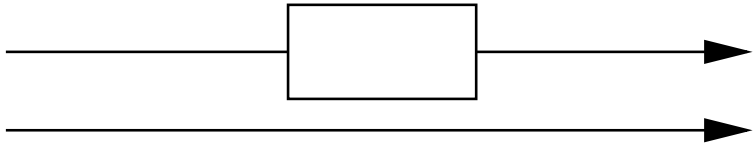
pure functions



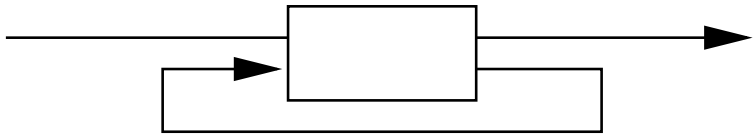
composition



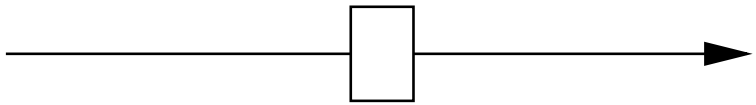
bypass



feedback

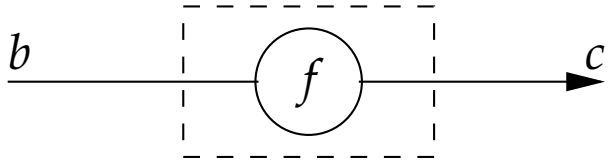


primitive components

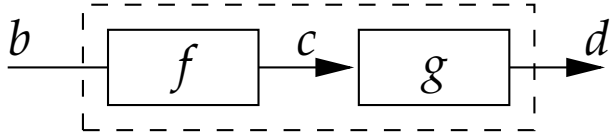


Nooled

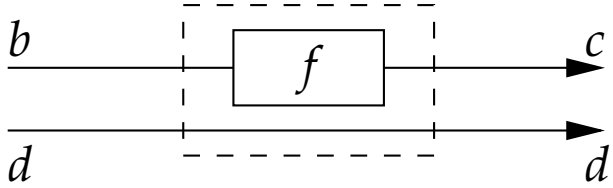
```
class Arrow a where
    pure :: (b -> c) -> a b c
```




```
    (>>>) :: a b c -> a c d -> a b d
```

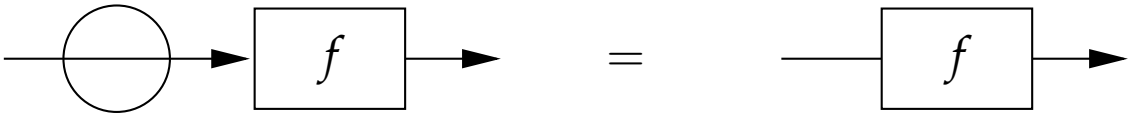



```
    first :: a b c -> a (b,d) (c,d)
```

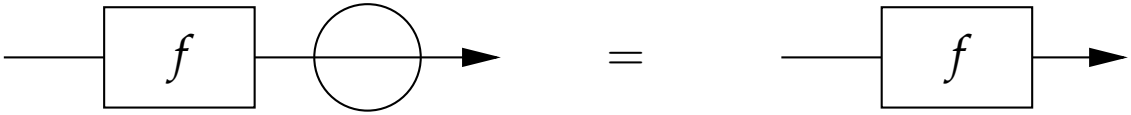


Noolte aksioomid

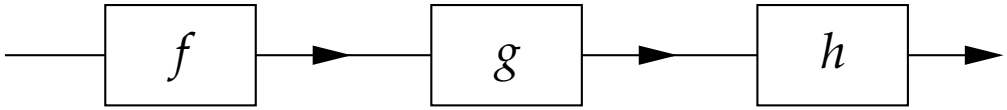
- `pure id >>> f = f`



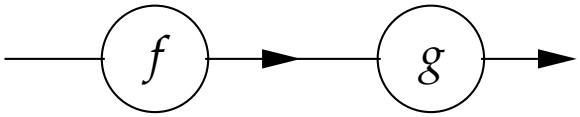
- `f >>> pure id = f`



- `(f >>> g) >>> h = f >>> (g >>> h)`



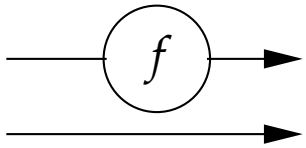
- `pure (g . f) = pure f >>> pure g`



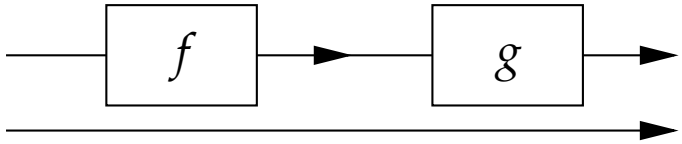
Noolte aksioomid

- `first (pure f)  =  pure (f |*| id)`

  `where f |*| g = \(x,y) -> (f x, g y)`



- `first (f >>> g)  =  first f >>> first g`

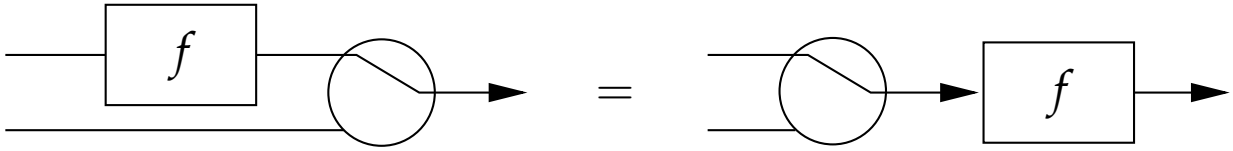


- `first f >>> pure (id |*| g)`

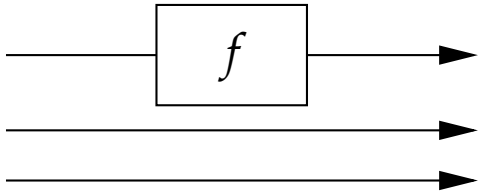
  `= pure (id |*| g) >>> first f`

Noolte aksioomid

- `first f >>> pure fst = pure fst >>> f`

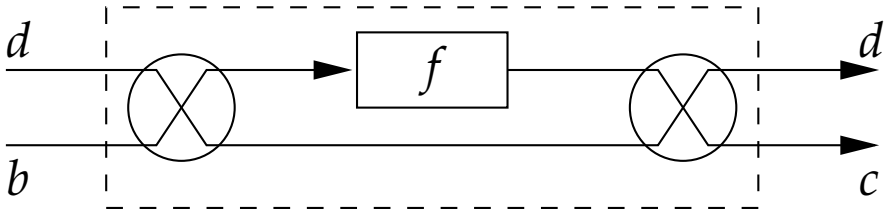


- `first (first f) >>> pure assoc`
  `= pure assoc >>> first f`
  `where assoc ((x,y),z) = (x,(y,z))`



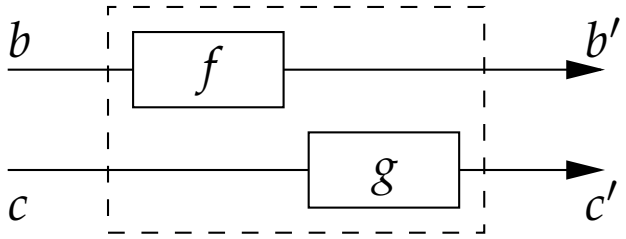
Noolte kombinaatoreid

```
arr     :: Arrow a => (b -> c) -> a b c
arr = pure

returnA :: Arrow a => a b b
returnA = arr id

second  :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
        where   swap ~(x,y) = (y,x)
```

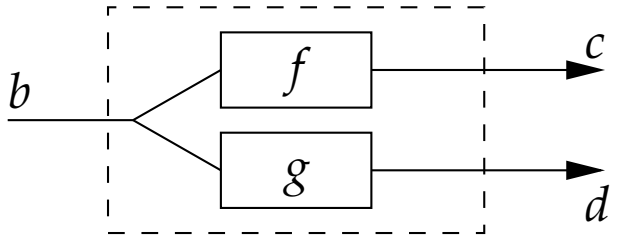


Noolte kombinaatoreid

```
(***) :: Arrow a => a b c -> a b' c' -> a (b,b') (c,c')
f *** g = first f >>> second g
```

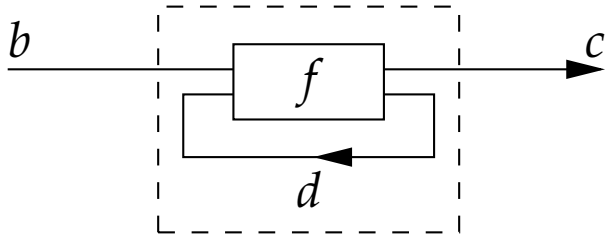



```
(&&&) :: Arrow a => a b c -> a b d -> a b (c,d)
f &&& g = arr (\b -> (b,b)) >>> f *** g
```



Tsüklitega nooled

```
class Arrow a => ArrowLoop a where
    loop :: a (b,d) (c,d) -> a b c
```



Konkreetseid nooli

- Funktsioonid

```
instance Arrow (->) where
    arr f = f
    f >>> g = g . f
    first f = \ ~(x,y) -> (f x, y)

trace :: ((b,d) -> (c,d)) -> b -> c
trace f b = let (c,d) = f (b,d) in c

instance ArrowLoop (->) where
    loop = trace
```

Konkreetseid nooli

- Olekuteisendajad

```
newtype State s i o = ST ((s,i) -> (s,o))

instance Arrow (State s) where
   pure f         = ST (id |*| f)
   ST f >>> ST g = ST (g . f)
   first (ST f)   = ST (assoc . (f |*| id) . unassoc)

instance ArrowLoop (->) where
   loop (ST f) = ST (trace (unassoc . f . assoc))
```

Konkreetseid nooli

- Monaadid

```
newtype Kleisli m a b = Kleisli (a -> m b)

instance Monad m => Arrow (Kleisli m) where
    arr f = Kleisli (return . f)
    Kleisli f >>> Kleisli g = Kleisli (\b -> f b >>= g)
    first (Kleisli f) = Kleisli (\ ~(b,d) -> f b >>= \c ->
                                             return (c,d))

instance MonadFix m => ArrowLoop (Kleisli m) where
    loop (Kleisli f) = Kleisli (liftM fst . mfix . f')
        where   f' x y = f (x, snd y)
```

Konkreetseid nooli

- Monaadid

```
newtype Kleisli m a b = Kleisli (a -> m b)

instance Monad m => Arrow (Kleisli m) where
    arr f = Kleisli (return . f)
    Kleisli f >>> Kleisli g = Kleisli (\b -> f b >>= g)
    first (Kleisli f) = Kleisli (\ ~(b,d) -> f b >>= \c ->
                                             return (c,d))

instance MonadFix m => ArrowLoop (Kleisli m) where
    loop (Kleisli f) = Kleisli (liftM fst . mfix . f')
        where   f' x y = f (x, snd y)
```

Konkreetseid nooli

- Striimprotsessorid

```
data Stream a = Cons a (Stream a)

zipStr :: (Stream a, Stream b) -> Stream (a,b)
zipStr (Cons x xs, Cons y ys) = Cons (x,y) (zipStr (xs,ys))

unzipStr :: Stream (a,b) -> (Stream a, Stream b)
unzipStr (Cons (x,y) xys) = (Cons x xs, Cons y ys)
   where (xs,ys) = unzipStr xys

instance Functor Stream where
   fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

Konkreetseid nooli

- Striimprotsessorid (järg)

```
newtype StrProc b c = SP (Stream b -> Stream c)

instance Arrow StrProc where
   pure f = SP (fmap f)
   SP f >>> SP g = SP (g . f)
   first (SP f) = SP (zipStr . (f |*| id) . unzipStr)

instance ArrowLoop StrProc where
   loop (SP f) = SP (loop (unzipStr . f . zipStr))
```

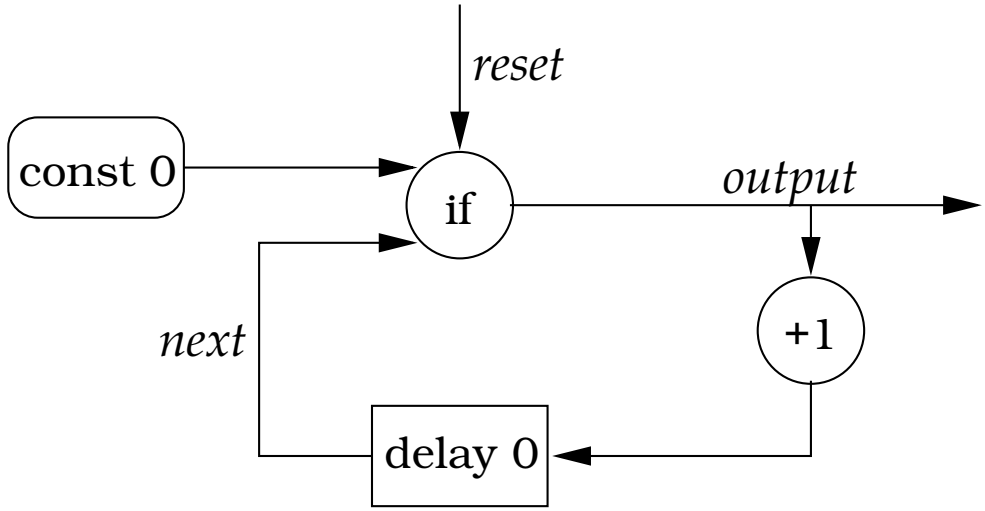
Konkreetseid nooli

- Viivitusega nooled

```
class ArrowLoop a => ArrowCircuit a where
   delay :: b -> a b b

instance ArrowCircuit StrProc where
   delay b = SP (Cons b)
```

Konkreetseid nooli

- Näide: alglaetav londur




```
counter :: ArrowCircuit a => a Bool Int
counter = loop (pure cond >>> pure dup >>>
                second (pure (+1) >>> delay 0))
    where cond (reset,next) = if reset then 0 else next
          dup x = (x,x)
```

Konkreetseid nooli

- Automaadid

```
newtype Auto i o = A (i -> (o,Auto i o))

instance Arrow Auto where
   pure f      = A (\b -> (f b, pure f))
   A f >>> A g = A (\b -> let (c,f') = f b
                             (d,g') = g c
                         in (d, f' >>> g'))
   first (A f) = A (\(b,d) -> let (c,f') = f b
                              in ((c,d), first f'))
instance ArrowLoop Auto where
   loop (A f) = A (\b -> let (~(c,d),f') = f (b,d)
                         in (c, loop f'))
instance ArrowCircuit Auto where
   delay b = A (\b' -> (b, delay b'))
```

Noolte notatsioon

- Noolte süntaks

$$
\begin{array}{lcl}
\mathit{exp} & = & \ldots \\
 & | & \textbf{proc}\ \mathit{pat}\ \texttt{->}\ \mathit{cmd} \\
\mathit{cmd} & = & \mathit{exp}\ \texttt{-<}\ \mathit{exp} \\
 & | & \textbf{do}\ \{\ \mathit{stmt};\ \ldots;\ \mathit{stmt};\ \mathit{cmd}\ \} \\
\mathit{stmt} & = & \mathit{pat}\ \texttt{<-}\ \mathit{cmd} \\
 & | & \mathit{cmd} \\
 & | & \textbf{rec}\ \{\ \mathit{stmt};\ \ldots;\ \mathit{stmt}\ \}
\end{array}
$$

Noolte notatsioon

- Transleerimisreeglid

$$\textbf{proc}\ p\ \texttt{->}\ f\ \texttt{-<}\ a = \begin{cases} \texttt{pure}(\lambda p \to a)\ \texttt{>>>}\ f & \text{if } FV(p) \cap FV(f) = \emptyset \\ \texttt{pure}(\lambda p \to (f, a))\ \texttt{>>>}\ \texttt{app} & \text{otherwise} \end{cases}$$

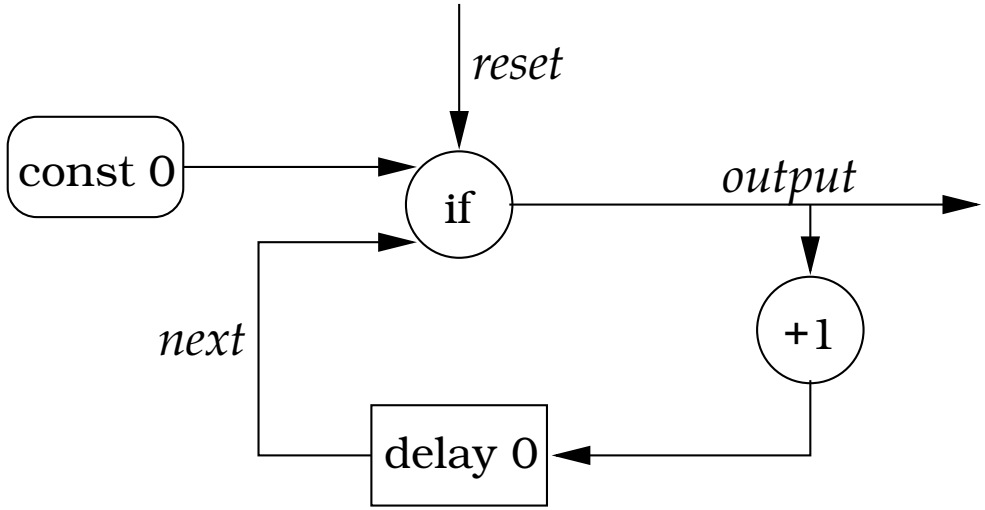
$$\textbf{proc}\ p\ \texttt{->}\ \textbf{do}\ \{\ c\ \} = \textbf{proc}\ p\ \texttt{->}\ c$$

$$\textbf{proc}\ p\ \texttt{->}\ \textbf{do}\ \{\ p'\ \texttt{<-}\ c;\ B\ \} = ((\textbf{proc}\ p\ \texttt{->}\ c)\ \texttt{\&\&\&}\ \texttt{returnA})\ \texttt{>>>}\ \textbf{proc}\ (p', p)\ \texttt{->}\ \textbf{do}\ \{\ B\ \}$$

$$\textbf{proc}\ p\ \texttt{->}\ \textbf{do}\ \{\ c;\ B\ \} = \textbf{proc}\ p\ \texttt{->}\ \textbf{do}\ \{\ \_\ \texttt{<-}\ c;\ B\ \}$$

$$\textbf{proc}\ p\ \texttt{->}\ \textbf{do}\ \{\ \textbf{rec}\ \{\ A\ \};\ B\ \} = \texttt{returnA}\ \texttt{\&\&\&}\ \texttt{loop}(\textbf{proc}\ (p, p_A)\ \texttt{->}\ \textbf{do}\ \{\ A;\ \texttt{returnA}\ \texttt{-<}\ (p_B, p_A)\})\ \texttt{>>>}\ \textbf{proc}\ (p, p_B)\ \texttt{->}\ \textbf{do}\ \{\ B\ \}$$

Alglaetav londur




```
counter :: ArrowCircuit a => a Bool Int
counter = proc reset -> do
    rec output <- returnA -< if reset then 0 else next
        next   <- delay 0 -< output + 1
    returnA -< output
```