This Lecture

- Monads in Haskell
- Some standard monads
- Combining effects: monad transformers
- Arrows
- FRP and Yampa
In Haskell, the notion of a monad is captured by a **Type Class**:

```haskell
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

Allows names of the common functions to be overloaded and sharing of derived definitions.
instance Monad Maybe where

    -- return :: a -> Maybe a
    return = Just

    -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing >>= _ = Nothing
    (Just x) >>= f = f x
Exercise 1: A State Monad in Haskell

Haskell 98 does not permit type synonyms to be instances of classes. Hence we have to define a new type:

```haskell
def newtype S a = S (Int -> (a, Int))
```

```haskell
unS :: S a -> (Int -> (a, Int))
unS (S f) = f
```

Provide a `Monad` instance for `S`.
Exercise 1: Solution

instance Monad S where
  return a = S (\s -> (a, s))

  m >>= f = S $ \s ->
  let (a', s') = unS m s
  in unS (f a) s'
To be useful, monads need to be equipped with additional operations specific to the effects in question. For example:

```haskell
fail :: String -> Maybe a
fail s = Nothing

catch :: Maybe a -> Maybe a -> Maybe a
m1 `catch` m2 =
  case m1 of
    Just _ -> m1
    Nothing -> m2
```
Typical operations on a state monad:

\[
\text{set} :: \text{Int} \rightarrow S () \\
\text{set } a = S (\_ \rightarrow ((), a))
\]

\[
\text{get} :: S \text{Int} \\
\text{get } = S (\_s \rightarrow (s, s))
\]

Moreover, need to “run” a computation. E.g.:

\[
\text{runS} :: S \text{a} \rightarrow \text{a} \\
\text{runS } m = \text{fst } (\text{unS } m 0)
\]
Haskell provides convenient syntax for programming with monads:

\[
\begin{align*}
do & \\
& a \leftarrow \text{exp}_1 \\
b & \leftarrow \text{exp}_2 \\
\text{return} & \text{exp}_3
\end{align*}
\]

is syntactic sugar for

\[
\begin{align*}
\text{exp}_1 & \gg= \lambda a \rightarrow \\
\text{exp}_2 & \gg= \lambda b \rightarrow \\
\text{return} & \text{exp}_3
\end{align*}
\]
The **do-notation** (2)

Computations can be done solely for effect, ignoring the computed value:

\[
\text{do} \\
\quad exp_1 \\
\quad exp_2 \\
\quad \text{return } exp_3
\]

is syntactic sugar for

\[
exp_1 \gg= \_ \rightarrow \\
exp_2 \gg= \_ \rightarrow \\
\text{return } exp_3
\]
The do-notation (3)

A let-construct is also provided:

\[
\begin{align*}
&\text{do} \\
&\text{let } a = \text{expr}_1 \\
&\quad b = \text{expr}_2 \\
&\quad \text{return } \text{expr}_3 \\
\end{align*}
\]

is equivalent to

\[
\begin{align*}
&\text{do} \\
&\quad a \gets \text{return } \text{expr}_1 \\
&\quad b \gets \text{return } \text{expr}_2 \\
&\quad \text{return } \text{expr}_3 \\
\end{align*}
\]
Numbering Trees in \texttt{do}-notation

\begin{verbatim}
numberTree :: Tree a \rightarrow Tree Int
numberTree t = runS (ntAux t)

where

ntAux :: Tree a \rightarrow S (Tree Int)
ntAux (Leaf _) = do
    n <- get
    set (n + 1)
    return (Leaf n)

ntAux (Node t1 t2) = do
    t1' <- ntAux t1
    t2' <- ntAux t2
    return (Node t1' t2')
\end{verbatim}
The Compiler Fragment Revisited (1)

Given a suitable “Diagnostics” monad $D$ that collects error messages, `enterVar` can be turned from this:

```
enterVar :: Id -> Int -> Type -> Env
        -> Either Env ErrorMgs
```

into this:

```
enterVarD :: Id -> Int -> Type -> Env
         -> D Env
```

and then `identDefs` from this . . .
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
where
  (e', ms1) = identAux l env e
  (env'', ms2) =
    case enterVar i l t env of
      Left env' -> (env', [],)
      Right m   -> (env, [m])
  (ds'', env''', ms3) =
    identDefs l env' ds
into this:

```haskell
identDefsD l env [] = return ([], env)
identDefsD l env ((i,t,e) : ds) = do
    e' <- identAuxD l env e
    env' <- enterVarD i l t env
    (ds'', env''') <- identDefsD l env' ds
    return ([(i,t,e') : ds'', env'''])
```

(Suffix D just to remind us the types have changed.)
The Compiler Fragment Revisited (4)

Compare with the “core” identified earlier!

\[
\begin{align*}
\text{identDefs } l \text{ env } [] & = ([], \text{ env}) \\
\text{identDefs } l \text{ env } ((i,t,e) : ds) & = \\
& ((i,t,e') : ds', \text{ env''}) \\
\text{where} & \\
\quad e' & = \text{identAux } l \text{ env } e \\
\quad \text{env'} & = \text{enterVar } i \text{ l t env} \\
(ds', \text{ env''}) & = \text{identDefs } l \text{ env'} \text{ ds}
\end{align*}
\]

The monadic version is very close to ideal, without sacrificing functionality, clarity, or pureness!
The List Monad

Computation with many possible results, "nondeterminism":

```
instance Monad [] where
    return a = [a]
    m >>= f = concat (map f m)
    fail s  = []
```

Example:
```
x <- [1, 2]
y <- ['a', 'b']
return (x,y)
```

Result:
```
[(1,'a'),(1,'b'),
(2,'a'),(2,'b')]
```
The Reader Monad

Computation in an environment:

\[
\text{instance Monad } ((\rightarrow) \ e) \text{ where}
\]
\[
\begin{align*}
\text{return } a &= \text{const } a \\
\text{m } \gg= f &= \lambda e \rightarrow f (m e) e
\end{align*}
\]

\[
\begin{align*}
\text{getEnv} :: ((\rightarrow) \ e) \ e \\
\text{getEnv} &= \text{id}
\end{align*}
\]
The Haskell IO Monad

In Haskell, IO is handled through the IO monad. IO is *abstract*! Conceptually:

```haskell
newtype IO a = IO (World -> (a, World))
```

Some operations:

- `putChar :: Char -> IO ()`
- `putStr :: String -> IO ()`
- `putStrLn :: String -> IO ()`
- `getChar :: IO Char`
- `getLine :: IO String`
- `getContents :: String`
Monad Transformers (1)

What if we need to support more than one type of effect?
What if we need to support more than one type of effect?

For example: State and Error/Partiality?
Monad Transformers (1)

What if we need to support more than one type of effect?

For example: State and Error/Partiality?

We could implement a suitable monad from scratch:

```
newtype SE s a = SE (s -> Maybe (a, s))
```
However:
Monad Transformers (2)

However:

- Not always obvious how: e.g., should the combination of state and error have been

  ```haskell
  newtype SE s a = SE (s -> (Maybe a, s))
  ```
Monad Transformers (2)

However:

- Not always obvious how: e.g., should the combination of state and error have been

  ```haskell
  newtype SE s a = SE (s -> (Maybe a, s))
  ```

- Duplication of effort: similar patterns related to specific effects are going to be repeated over and over in the various combinations.
Monad Transformers can help:
Monad Transformers can help:

- A *monad transformer* transforms a monad by adding support for an additional effect.
Monad Transformers (3)

*Monad Transformers* can help:

- A *monad transformer* transforms a monad by adding support for an additional effect.
- A library of monad transformers can be developed, each adding a specific effect (state, error, ...), allowing the programmer to mix and match.
Monad Transformers can help:

- A *monad transformer* transforms a monad by adding support for an additional effect.
- A library of monad transformers can be developed, each adding a specific effect (state, error, ...), allowing the programmer to mix and match.
- A form of *aspect-oriented programming*.
A **monad transformer** maps monads to monads. Represented by a type constructor $T$ of the following kind:

$$T : : (\ast \rightarrow \ast) \rightarrow (\ast \rightarrow \ast)$$
• A *monad transformer* maps monads to monads. Represented by a type constructor $T$ of the following kind:

$$T :: (\star \to \star) \to (\star \to \star)$$

• Additionally, a monad transformer *adds* computational effects. A mapping $lift$ from computations in the underlying monad to computations in the transformed monad is needed:

$$lift :: M a \to T M a$$
• These requirements are captured by the following (multi-parameter) type class:

```haskell
class (Monad m, Monad (t m)) => MonadTransformer t m where
  lift :: m a -> t m a
```
A monad transformer adds specific effects to any monad. Thus the effect-specific operations needs to be overloaded. For example:

```haskell
class Monad m => E m where
    eFail :: m a
    eHandle :: m a -> m a -> m a

class Monad m => S m s | m -> s where
    sSet :: s -> m ()
    sGet :: m s
```
The Identity Monad

We are going to construct monads by successive transformations of the identity monad:

```haskell
newtype I a = I a
unI (I a) = a

instance Monad I where
    return a = I a
    m >>= f = f (unI m)

runI :: I a -> a
runI = unI
```
newtype ET m a = ET (m (Maybe a))
unET (ET m) = m

Any monad transformed by \texttt{ET} is a monad:

instance Monad m => Monad (ET m) where
    return a = ET (return (Just a))

    m >>= f = ET $ do
        ma <- unET m
        case ma of
            Nothing -> return Nothing
            Just a  -> unET (f a)
The Error Monad Transformer (2)

We need the ability to run transformed monads:

```haskell
runET :: Monad m => ET m a -> m a
runET etm = do
    ma <- unET etm
    case ma of
        Just a    -> return a
        Nothing   -> error "Should not happen"
```

**ET** is a monad transformer:

```haskell
instance Monad m => MonadTransformer ET m where
    lift m = ET (m >>= \a -> return (Just a))
```
Any monad transformed by \( ET \) is an instance of \( E \):

\[
\text{instance Monad m => E (ET m) where}
\]
\[
\begin{align*}
\text{eFail} & = ET \ (\text{return Nothing}) \\
\text{m1 \ 'eHandle' \ m2} & = ET \ \_ \ do \\
& \quad \text{ma <- unET m1} \\
& \quad \text{case ma of} \\
& \quad \text{Nothing} \rightarrow \text{unET m2} \\
& \quad \text{Just _} \rightarrow \text{return ma}
\end{align*}
\]
The Error Monad Transformer (4)

A state monad transformed by $\text{ET}$ is a state monad:

\[
\text{instance } S \ m \ s \Rightarrow S \ (\text{ET} \ m) \ s \ \text{where}
\]
\[
\begin{align*}
    \text{sSet } s &= \text{lift } (\text{sSet } s) \\
    \text{sGet} &= \text{lift } \text{sGet}
\end{align*}
\]
Let

\[
\text{ex2} = \text{eFail} \ '\text{eHandle}' \ \text{return} \ 1
\]

1. Suggest a possible type for \text{ex2}.
   \text{(Assume} \ 1 :: \text{Int}.)

2. Given your type, use the appropriate combination of “run functions” to run \text{ex2}.
Exercise 2: Solution

```
ex2 :: ET I Int
ex2 = eFail 'eHandle' return 1

ex2result :: Int
ex2result = runI (runET ex2)
```
The State Monad Transformer (1)

newtype ST s m a = ST (s -> m (a, s))
unST (ST m) = m

Any monad transformed by \texttt{ST} is a monad:

\texttt{instance Monad m} => Monad (ST s m) \texttt{where}
return a = ST (\s -> return (a, s))
m >>= f = ST $ \s -> do
(a, s') <- unST m s
unST (f a) s'

LiU-FP2010 Part II: Lecture 6 -- p.33/83
We need the ability to run transformed monads:

```haskell
runST :: Monad m => ST s m a -> s -> m a
runST stf s0 = do
    (a, _) <- unST stf s0
    return a
```

**ST is a monad transformer:**

```haskell
instance Monad m => MonadTransformer (ST s) m where
    lift m = ST (\s -> m >>= \a ->
        return (a, s))
```
The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

```hs
instance Monad m => S (ST s m) s where
    sSet s = ST (\_ -> return ((), s))
    sGet  = ST (\s -> return (s, s))
```

An error monad transformed by ST is an error monad:

```hs
instance E m => E (ST s m) where
    eFail = lift eFail
    m1 `eHandle` m2 = ST $ \s ->
        unST m1 s `eHandle` unST m2 s
```
Exercise 3: Effect Ordering

Consider the code fragment

```haskell
ex3a :: (ST Int (ET I)) Int
ex3a = (sSet 42 >> eFail) `eHandle` sGet
```

Note that the exact same code fragment also can be typed as follows:

```haskell
ex3b :: (ET (ST Int I)) Int
ex3b = (sSet 42 >> eFail) `eHandle` sGet
```

What is

```haskell
runI (runET (runST ex3a 0))
runI (runST (runET ex3b) 0)
```
Exercise 3: Solution

\[
\begin{align*}
\text{runI (runET (runST ex3a 0))} &= 0 \\
\text{runI (runST (runET ex3b) 0)} &= 42
\end{align*}
\]

Why? Because:

\[
\begin{align*}
\text{ST s (ET I) a} &\equiv s \rightarrow (ET I) (a, s) \\
&\equiv s \rightarrow I (\text{Maybe} (a, s)) \\
&\equiv s \rightarrow \text{Maybe} (a, s)
\end{align*}
\]

\[
\begin{align*}
\text{ET (ST s I) a} &\equiv (ST s I) (\text{Maybe a}) \\
&\equiv s \rightarrow I (\text{Maybe a, s}) \\
&\equiv s \rightarrow (\text{Maybe a, s})
\end{align*}
\]
Exercise 4: Alternative \( \texttt{ST} \)?

To think about.

Could \( \texttt{ST} \) have been defined in some other way, e.g.

\[
\text{newtype } \texttt{ST} \ s \ m \ a = \texttt{ST} \ (m \ (s \to \ (a, s)))
\]

or perhaps

\[
\text{newtype } \texttt{ST} \ s \ m \ a = \texttt{ST} \ (s \to \ (m \ a, s))
\]
Problems with Monad Transformers

- With one transformer for each possible effect, we get a lot of combinations: the number grows quadratically; each has to be instantiated explicitly.

- Jaskelioff (2008, 2009) has proposed a possible, more extensible alternative.
System descriptions in the form of block diagrams are very common. Blocks have inputs and outputs and can be combined into larger blocks. For example, serial composition:
System descriptions in the form of block diagrams are very common. Blocks have inputs and outputs and can be combined into larger blocks. For example, serial composition:

A *combinator* can be defined that captures this idea:

\[(\triangleright\triangleright\triangleright) \; :: \; B \ a \ b \rightarrow B \ b \ c \rightarrow B \ a \ c\]
But systems can be complex:
But systems can be complex:

How many and what combinators do we need to be able to describe arbitrary systems?
Arrows (3)

John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types (or “blocks”, if you prefer).
John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types (or “blocks”, if you prefer).
- Particularly suitable for types representing process-like computations.
Arrows (3)

John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types (or “blocks”, if you prefer).
- Particularly suitable for types representing process-like computations.
- Related to *monads*, since arrows are computations, but more general.
John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types (or “blocks”, if you prefer).
- Particularly suitable for types representing process-like computations.
- Related to *monads*, since arrows are computations, but more general.
- Provides a minimal set of “wiring” combinators.
What is an arrow? (1)

- A *type constructor* of arity two.
What is an arrow? (1)

- A *type constructor* $a$ of arity two.
- Three operators:
What is an arrow? (1)

- A *type constructor* `a` of arity two.
- Three operators:
  - *lifting*:
    
    \[
    \text{arr} :: (b \rightarrow c) \rightarrow a \ b \ c
    \]
What is an arrow? (1)

- A type constructor \( \text{a} \) of arity two.
- Three operators:
  - lifting:
    \[
    \text{arr} :: (b \rightarrow c) \rightarrow \text{a} b c
    \]
  - composition:
    \[
    (\gg\gg\gg) :: \text{a} b c \rightarrow \text{a} c d \rightarrow \text{a} b d
    \]
What is an arrow? (1)

- A **type constructor** \( a \) of arity two.
- Three operators:
  - **lifting**:
    \[
    \text{arr} :: (b \to c) \to a \ b \ c
    \]
  - **composition**:
    \[
    \text{>>>(}) :: a \ b \ c \to a \ c \ d \to a \ b \ d
    \]
  - **widening**:
    \[
    \text{first} :: a \ b \ c \to a \ (b,d) \ (c,d)
    \]
What is an arrow? (1)

- A type constructor `a` of arity two.
- Three operators:
  - **lifting**:
    \[
    \text{arr} :: (b \to c) \to a \ b \ c
    \]
  - **composition**:
    \[
    (\gg\gg\gg) :: a \ b \ c \to a \ c \ d \to a \ b \ d
    \]
  - **widening**:
    \[
    \text{first} :: a \ b \ c \to a \ (b,d) \ (c,d)
    \]
- A set of **algebraic laws** that must hold.
What is an arrow? (2)

These diagrams convey the general idea:

\[ \text{arr } f \rightarrow f \rightarrow g \]

\[ \text{first } f \rightarrow f \rightarrow g \]
The **Arrow class**

In Haskell, a **type class** is used to capture these ideas (except for the laws):

```haskell
class Arrow a where
    arr :: (b -> c) -> a b c
    (>>>) :: a b c -> a c d -> a b d
    first :: a b c -> a (b,d) (c,d)
```
Functions are arrows (1)

Functions are a simple example of arrows, with \((-\to)\) as the arrow type constructor.

**Exercise 5:** Suggest suitable definitions of

- `arr`
- `(>>>)`
- `first`

for this case!

(We have not looked at what the laws are yet, but they are “natural”.)
Functions are arrows (2)

Solution:

- arr = id
Solution:

- \( \text{arr} = \text{id} \)

To see this, recall

\[
\text{id} :: t \rightarrow t \\
\text{arr} :: (b \rightarrow c) \rightarrow a \ b \ c
\]
Functions are arrows (2)

Solution:

- \textit{arr} = \textit{id}

To see this, recall

\textit{id} :: t \rightarrow t

\textit{arr} :: (b \rightarrow c) \rightarrow a \ b \ c

Instantiate with

\[ a = (\rightarrow) \]

\[ t = b \rightarrow c = (\rightarrow) \ b \ c \]
Functions are arrows (3)

- $f >>> g = \lambda a \rightarrow g (f a)$
Functions are arrows (3)

- \( f >>> g = \lambda a \rightarrow g (f a) \)  \text{ or }  
- \( f >>> g = g \cdot f \)
Functions are arrows (3)

- $f >>> g = \lambda a \rightarrow g \ (f \ a) \quad or$
- $f >>> g = g \ . \ f \quad or \ even$
- $(>>>) = flip \ (.)$
Functions are arrows (3)

- \( f >>> g = \lambda a \rightarrow g \ (f \ a) \) \ or
- \( f >>> g = g \ . \ f \) \ or even
- \( (>>>) = \text{flip} \ (.) \)
- \( \text{first} \ f = \lambda (b,d) \rightarrow (f \ b, d) \)
Arrow instance declaration for functions:

```haskell
instance Arrow (->) where
  arr        = id
  (>>>)      = flip (.)
  first f = \(b,d) -> (f b, d)
```

Functions are arrows (4)
Some arrow laws

\[(f \gggg g) \gggg h = f \gggg (g \gggg h)\]
Some arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr } (f >>> g) = \text{arr } f >>> \text{arr } g\]
Some arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr } (f >>> g) = \text{arr } f >>> \text{arr } g\]
\[\text{arr } \text{id} >>> f = f\]
Some arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]

\[\text{arr} (f >>> g) = \text{arr } f >>> \text{arr } g\]

\[\text{arr id} >>> f = f\]

\[f = f >>> \text{arr id}\]
Some arrow laws

\[(f \>>> g) \>>> h = f \>>> (g \>>> h)\]
\[\operatorname{arr} (f \>>> g) = \operatorname{arr} f \>>> \operatorname{arr} g\]
\[\operatorname{arr} \operatorname{id} \>>> f = f\]
\[f = f \>>> \operatorname{arr} \operatorname{id}\]
\[\operatorname{first} \left(\operatorname{arr} f \right) = \operatorname{arr} \left(\operatorname{first} f \right)\]
Some arrow laws

\[(f \ggg g) \ggg h = f \ggg (g \ggg h)\]

\[\text{arr } (f \ggg g) = \text{arr } f \ggg \text{arr } g\]

\[\text{arr } \text{id} \ggg f = f\]

\[f = f \ggg \text{arr } \text{id}\]

\[\text{first } (\text{arr } f) = \text{arr } (\text{first } f)\]

\[\text{first } (f \ggg g) = \text{first } f \ggg \text{first } g\]
Another important operator is \texttt{loop}: a fixed-point operator used to express recursive arrows or \textit{feedback}:

\[
\text{loop } f
\]
The loop combinator (2)

Not all arrow instances support \texttt{loop}. It is thus a method of a separate class:

\begin{verbatim}
class Arrow a => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c
\end{verbatim}

Remarkably, the four combinators \texttt{arr}, \texttt{>>>}, \texttt{first}, and \texttt{loop} are sufficient to express any conceivable wiring!
Some more arrow combinators (1)

second :: Arrow a => a b c -> a (d, b) (d, c)

(*** *) :: Arrow a => a b c -> a d e -> a (b, d) (c, e)

(&&&) :: Arrow a => a b c -> a b d -> a b (c, d)
Some more arrow combinators (2)

As diagrams:

second $f$

$f$ $\&\&\&$ $g$

$f$ $\&\&\&$ $g$
Some more arrow combinators (3)
Some more arrow combinators (3)

second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)
Some more arrow combinators (3)

second :: Arrow a => a b c → a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)

(*** ) :: Arrow a =>
    a b c → a d e → a (b,d) (c,e)
f *** g = first f >>> second g
Some more arrow combinators (3)

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

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

(&&&) :: Arrow a => a b c -> a b d -> a b (c,d)
f &&& g = arr (\x->(x,x)) >>> (f *** g)
Exercise 6

Describe the following circuit using arrow combinators:

\[ a_1, a_2, a_3 :: \text{A Double Double} \]
Exercise 3: Describe the following circuit using arrow combinators:

\[ a1, a2, a3 :: A \text{ Double Double} \]
Exercise 3: Describe the following circuit using arrow combinators:

\[
\text{a1, a2, a3 :: A Double Double}
\]

\[
\text{circuit_v1 :: A Double Double}
\]

\[
\text{circuit_v1 = (a1 &&& arr id)}
\]

\[
\text{>>> (a2 *** a3)}
\]

\[
\text{>>> arr (uncurry (+))}
\]
Exercise 3: Describe the following circuit:

\[ a_1, a_2, a_3 :: \text{A Double Double} \]
Exercise 3: Describe the following circuit:

\[
\begin{align*}
\text{a1, a2, a3} & : \text{A Double Double} \\
\text{circuit\_v2} & : \text{A Double Double} \\
\text{circuit\_v2} & = \text{arr } (\lambda x \rightarrow (x,x)) \\
& \quad \triangleright\triangleright\triangleright \text{ first a1} \\
& \quad \triangleright\triangleright\triangleright (\text{a2 } *** \text{ a3}) \\
& \quad \triangleright\triangleright\triangleright \text{ arr (uncurry (+))}
\end{align*}
\]
The arrow \texttt{do} notation (1)

Ross Paterson’s \texttt{do}-notation for arrows supports \textit{pointed} arrow programming. Only \textit{syntactic sugar}.

\begin{verbatim}
proc pat -> do [ rec ]
pat_1 <- sfexp_1 <- exp_1
pat_2 <- sfexp_2 <- exp_2
...
pat_n <- sfexp_n <- exp_n
returnA <- exp
\end{verbatim}

\textbf{Also:} \texttt{let pat = exp} \equiv \texttt{pat <- arr id <- exp}
The arrow do notation (2)

Let us redo exercise 3 using this notation:

```
circuit_v4 :: A Double Double
circuit_v4 = proc x -> do
    y1 <- a1 <+ x
    y2 <- a2 <+ y1
    y3 <- a3 <+ x
    returnA <+ y2 + y3
```
The arrow do notation (3)

We can also mix and match:

circuit_v5 :: A Double Double

circuit_v5 = proc x -> do
  y2 <- a2 <<< a1 --< x
  y3 <- a3 --< x
  returnA --< y2 + y3
The arrow do notation (4)

Recursive networks: do-notation:

\[ a_1, a_2 :: A \text{ Double Double} \]
\[ a_3 :: A \text{ (Double,Double) Double} \]
The arrow do notation (4)

Recursive networks: do-notation:

```
a1, a2 :: A Double Double
a3 :: A (Double,Double) Double
```

**Exercise 5:** Describe this using only the arrow combinators.
The arrow do notation (5)

circuit = proc x -> do
  rec
    y1 <- a1 <- x
    y2 <- a2 <- y1
    y3 <- a3 <- (x, y)
  let y = y2 + y3
  returnA <- y
Arrows and Monads (1)

Arrows generalize monads: for every monad type there is an arrow, the *Kleisli category* for the monad:

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

instance Monad m => Arrow (Kleisli m) where
  arr f = K (\b -> return (f b))
  K f >>> K g = K (\b -> f b >>= g)
```
But not every arrow is a monad. However, arrows that support an additional apply operation are effectively monads:

\[
\text{apply} :: \text{Arrow } a \Rightarrow a \ (a \ b \ c, \ b) \ c
\]

Exercise 7: Verify that

\[
\text{newtype } M \ b = M \ (A \ () \ b)
\]

is a monad if \( A \) is an arrow supporting apply; i.e., define return and bind in terms of the arrow operations (and verify that the monad laws hold).
An application: FRP

Functional Reactive Programming (FRP):

- Paradigm for *reactive programming* in a functional setting:
  - Input arrives *incrementally* while system is running.
  - Output is generated in response to input in an interleaved and *timely* fashion.
An application: FRP

Functional Reactive Programming (FRP):

- Paradigm for *reactive programming* in a functional setting:
  - Input arrives *incrementally* while system is running.
  - Output is generated in response to input in an interleaved and *timely* fashion.

- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
Functional Reactive Programming (FRP):

- Paradigm for *reactive programming* in a functional setting:
  - Input arrives *incrementally* while system is running.
  - Output is generated in response to input in an interleaved and *timely* fashion.

- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).

- Has evolved in a number of directions and into different concrete implementations.
Yampa:

- The most recent Yale FRP implementation.
**Yampa**

*Yampa:*

- The most recent Yale FRP implementation.
- *Embedding* in Haskell (a Haskell library).
Yampa:

- The most recent Yale FRP implementation.
- *Embedding* in Haskell (a Haskell library).
- *Arrows* used as the basic structuring framework.
Yampa:

- The most recent Yale FRP implementation.
- *Embedding* in Haskell (a Haskell library).
- *Arrows* used as the basic structuring framework.
- *Continuous time*. 
Yampa

**Yampa:**

- The most recent Yale FRP implementation.
- *Embedding* in Haskell (a Haskell library).
- *Arrows* used as the basic structuring framework.
- *Continuous time*.
- Discrete-time signals modelled by continuous-time signals and an option type.
Yampa

**Yampa:**

- The most recent Yale FRP implementation.
- *Embedding* in Haskell (a Haskell library).
- *Arrows* used as the basic structuring framework.
- *Continuous time*.
- Discrete-time signals modelled by continuous-time signals and an option type.
- Advanced *switching constructs* allows for highly dynamic system structure.
Related languages

FRP related to:

- Synchronous languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink.
Related languages

FRP related to:

• Synchronous languages, like Esterel, Lucid Synchrone.
• Modeling languages, like Simulink.

Distinguishing features of FRP:

• First class reactive components.
• Allows highly dynamic system structure.
• Supports hybrid (mixed continuous and discrete) systems.
FRP applications

Some domains where FRP has been used:

- Graphical Animation (Fran: Elliott, Hudak)
- Robotics (Frob: Peterson, Hager, Hudak, Elliott, Pembeci, Nilsson)
- Vision (FVision: Peterson, Hudak, Reid, Hager)
- GUIs (Fruit: Courtney)
- Hybrid modeling (Nilsson, Hudak, Peterson)
Yampa?
Yampa?

Yet
Another
Mostly
Pointless
Acronym
Yampa?

Yet
Another
Mostly
Pointless
Acronym

???
Yampa?

Yet
Another
Mostly
Pointless
Acronym

???

No ...
Yampa?

Yampa is a river . . .
Yampa?

... with long calmly flowing sections ...
Yampa?

... and abrupt whitewater transitions in between.

A good metaphor for hybrid systems!
Signal functions

Key concept: *functions on signals.*
Signal functions

Key concept: *functions on signals*.

Intuition:

Signal $\alpha \approx \text{Time} \rightarrow \alpha$

$x :: \text{Signal } T1$

$y :: \text{Signal } T2$

$\text{SF } \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta$

$f :: \text{SF } T1 \text{ T2}$
Signal functions

Key concept: *functions on signals.*

Intuition:

\[
\text{Signal } \alpha \approx \text{Time} \rightarrow \alpha
\]

\[
x :: \text{Signal T1}
\]

\[
y :: \text{Signal T2}
\]

\[
\text{SF } \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta
\]

\[
f :: \text{SF T1 T2}
\]

Additionally: *causality* requirement.
Signal functions and state

Alternative view:
Signal functions and state

Alternative view:

Signal functions can encapsulate state.\( state(t) \) summarizes input history \( x(t'), t' \in [0, t] \).
Signal functions and state

Alternative view:

Signal functions can encapsulate state.

\( \text{state}(t) \) summarizes input history \( x(t'), t' \in [0, t] \).

Functions on signals are either:

- **Stateful**: \( y(t) \) depends on \( x(t) \) and \( \text{state}(t) \)
- **Stateless**: \( y(t) \) depends only on \( x(t) \)
Yampa and Arrows

SF is an arrow. Signal function instances of core combinators:

• \( \text{arr} :: (a \rightarrow b) \rightarrow SF\ a\ b \)
• \( \text{>>>} :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c \)
• \( \text{first} :: SF\ a\ b \rightarrow SF\ (a,c)\ (b,c) \)
• \( \text{loop} :: SF\ (a,c)\ (b,c) \rightarrow SF\ a\ b \)

But \text{apply} has no useful meaning. Hence \text{SF is not a monad.}
Some further basic signal functions

- `identity :: SF a a`
  `identity = arr id`
Some further basic signal functions

- `identity :: SF a a`
  
  `identity = arr id`

- `constant :: b -> SF a b`
  
  `constant b = arr (const b)`
Some further basic signal functions

- `identity :: SF a a`
  
  `identity = arr id`

- `constant :: b -> SF a b`
  
  `constant b = arr (const b)`

- `integral :: VectorSpace a s=>SF a a`
  
  `-`
Some further basic signal functions

- `identity :: SF a a`
  \[
  \text{identity} = \text{arr id}
  \]
- `constant :: b \rightarrow SF a b`
  \[
  \text{constant } b = \text{arr } (\text{const } b)
  \]
- `integral :: \text{VectorSpace } a \implies \text{SF } a a a`
- `time :: SF a Time`
  \[
  \text{time} = \text{constant } 1.0 >>> \text{integral}
  \]
Some further basic signal functions

- **identity** :: SF a a
  identity = arr id

- **constant** :: b -> SF a b
  constant b = arr (const b)

- **integral** :: VectorSpace a s=>SF a a

- **time** :: SF a a
  time = constant 1.0 >>> integral

- **(^<<) :: (b->c) -> SF a b -> SF a c**
  f (^<<) sf = sf >>> arr f
Example: A bouncing ball

\[ y = y_0 + \int v \, dt \]

\[ v = v_0 + \int -9.81 \]

On impact:

\[ v = -v(t-) \]

(fully elastic collision)
Part of a model of the bouncing ball

Free-falling ball:

type Pos = Double

type Vel = Double

fallingBall ::

    Pos -> Vel -> SF () (Pos, Vel)

fallingBall y0 v0 = proc () -> do

    v <- (v0 +) ^<< integral <- -9.81

    y <- (y0 +) ^<< integral <- v

    returnA <- (y, v)
**Dynamic system structure**

*Switching* allows the structure of the system to evolve over time:
Example: Space Invaders
Overall game structure

dpSwitch

route

ObjInput

ObjOutput

alien

gun

alien

bullet

[ObjectOutput]

killOrSpawn
Reading (1)


Reading (2)

Reading (3)


Reading (4)
